Computers For Smart People

Robert S. Swiatek

Robert S. Swiatek

First edition

ISBN: 0-9817843-9-9

available only as an ebook

also by Robert S. Swiatek

Don't Bet On It

Tick Tock, Don't Stop – A Manual For Workaholics

for seeing eye dogs only

This Page Intentionally Left Blank – Just Like The Paychecks Of The Workers

I Don't Want To Be A Pirate – Writer, maybe

wake up – it's time for your sleeping pill

Take Back The Earth – The Dumb, Greedy Incompetents Have Trashed It

Press 1 For Pig Latin

This War Won't Cost Much – I'm Already Against The Next One

here's your free gift – send $10 for shipping

Mirror, Mirror, On My Car

Save The Animals And Children

Recipes For Joy In Life

I’d like to thank all the people who made this book possible, in particular, all those people I met during my stay in Binghamton in the early 1970s. I especially thank my niece, Elizabeth Thomann-Stellrecht, who was responsible for the great cover of this book. Her work can be found on the vast majority of the books that I published since the spring of 2008 – that’s much more than a two-thirds majority needed to overcome Republican objections. Over the last few years, people have raved about these covers at various arts and crafts festivals of which I have been a part. Some have even purchased a book. Thanks!

To Pat and Lou

Table of contents

Introduction

I began writing my very first book in September 1972. It dealt with computer concepts and was meant as a high school math textbook to teach programming. It used APL, which stands for *A Programming Language*, a highly scientific language for the computer. At the time, a few publishing companies expressed interest in the book but as the days passed, they declined in getting the book on the market. I wasn't completely discouraged.

Their excuse was that there were enough of these types of books out there and I accepted that. At the same time I saw a dilemma insofar as books written about fairly common subject matter would not get printed for this same reason but revolutionary topics probably wouldn't make it to print either because the publisher wouldn't want to risk getting into an untested, unknown area. I never did submit it to a far-out press, even though this was just after Woodstock.

I did use the book when I taught a programming course in high school shortly thereafter, in addition to the regular APL textbook. However, once I left teaching the book was stored away gathering dust, rarely to be perused. Over time I realized that there was no chance that it would ever get published in its existing form. I also thought that it could be revised, with the original language of APL replaced by a common, understandable language. In this way it could have relevance. Of course, that meant almost a complete rewrite of the book.

In August 2001 on a Sunday afternoon I decided to dig out the book and redo it. I went through it but decided not to do it. The next day I changed my mind. I wound up revitalizing and resuscitating it using a generic language. This turned out to be a language that I created, utilizing features of many computer languages that I had come in contact with over the years. Since all languages do basically the same thing but by different means, I took all the benefits of each language and combined them into my language. The book would now be used to illustrate what computer programming is all about to people unfamiliar with the subject.

The intent of this book is to reach two types of people. The first are those who would like to get an idea of what programming is all about since that may be what they want to do as a profession. The other person to be reached is that individual who has little computer knowledge but would like some insight into what programming involves. This breakdown includes a great number of people.

By no means is this book meant to be a computer text but rather a means of spreading knowledge about computer programming. My goal is to make clear each topic presented but anyone reading the work need not feel disappointed if some area of the text is not completely comprehended. However, by the time someone is through with the book, it is my goal that either they will decide to pursue this field or at least have some basic understanding of what programming is all about.

Despite the possibility of getting this book published so many years ago when I first wrote it, there were a few things missing. As I mentioned, the language was too difficult for most readers to comprehend. Also, since it was my first book, it was missing what could be found in the books I wrote after it, namely at least a small amount of

humor. The subject matter may have limited that aspect, but as I have found, there are very few books where levity can't be interjected.

In general, it is probably better that the book didn't get published at that time. It really wasn't ready to come into print. However, when I revised it in 2001, all these limitations would be gone. Half a dozen years later, the work still wasn't published. I did some more modifications in January 2010 while staying in my cousin Jim's town home in Sun City Center, Florida. Incidentally, I have at least three cousins with that name. Then in December 2011, I decided to publish it as an ebook, resulting in a great deal more editing. Since I had created my own computer language, that created the biggest holdup. I felt for the longest time that the programs – few though they were – had to be thoroughly checked over since they couldn't really be tested with a computer. I needed to put in the effort to get this task done.

Somehow, I came up with a new idea. Every program found in this book is here for instructional purposes. It is meant to display computer concepts and who really cares if there are a few places where something may fail or could be done better. In reality, these programs have nowhere near the bugs that you will find on the Internet or even on your PC, each of which is rushed into production without checking. As you can tell, that approach was not done in this book. After all, quite a few years had passed since I started writing it. For that reason, any problems that you encounter in this work are truly minor and can easily be overlooked. If you are a person who likes to correct others by finding tiny mistakes in books, I need only remind you that every book that was ever written has at least one error, of some kind or another. Don't waste your time.

As far as the book title that I first chose, this goes back to my days at Binghamton University when I was studying for a degree in computer science. My fellow students and I worked together as a team to get projects done. The effort required was intense but we had a good sense of humor about it. In fact while going through the degree program one of my study-partners remarked, "Six months ago I could not spell computer programmer – now I are one!"

We all got a laugh out of that, and I loved that title. However, I decided that there wasn't enough room on the cover to put all those words – if I used a smaller font, not many people could read it – so I thought about another one that would be better. This didn't come easy, but eventually I settled on ***Computer For Smart People***. I hope you find this treatise to be enjoyable and enlightening.

1. Elements of language

Any language that we come in contact with follows certain rules. This applies to Spanish, English or any computer language. Naturally the fewer rules there are, the easier the language. As the number of rules increase, so does the difficulty. Unfortunately there may be no choice but to have a preponderance of rules, such as the language of a computer system. However, I shall get into that later.

For now, let us talk about the language of English, although you will soon realize that what applies here will be the same for any language we consider. We have to start with certain basic symbols, specifically the letters of the alphabet that apply to this language. In our case they're the letters *a* through *z*. But we also need to mention the capital letters, *A* through *Z* as well as certain punctuation, such as the comma, period, question mark and a few other symbols. I think you get the idea. Our character set will be around 75 different symbols. As we progress we shall be introduced to more and more of them.

These elements or basic symbols will be put together to form words. Thus the letters "t", "h" and "e" form the word, "the." Some letters put together may not form a valid word, such as "q", "j", "x" and "h," no matter what order we put them in. You might reply that you went to school with a guy from Russia whose name was exactly those letters in that same order, but that doesn't count. Some combinations will give us words while others may not. There could come a day when the four letters we mentioned form a valid word, since new words come into existence from time to time in the English language.

A few examples of words that feature symbols other than our usual letters of the alphabet are "son-in-law" and "o'clock," and just recently one of my favorites, "24/7." Thus we need the hyphen, apostrophe and the slash, since some words use these characters. We will run into situations where other symbols will be used, which will be discussed when needed.

You might ask how it is determined whether a combination of letters is a valid word. This is decided by predefined rules of the language. By referring to an English dictionary, you can see whether you have a word. If you care to read an appropriate book, check out ***The Professor and the Madman*** by Simon Winchester. It's a tale of murder, insanity as well as the making of the Oxford Dictionary.

As you can imagine, there are various editions of the dictionary as well as those produced by different companies. This will mean that one dictionary might indicate that one combination of letters is a word while another may not have it listed. This difference along with the additions and deletions of words to the dictionary with each passing day adds to the complexity of the language, which we will not be burdened with.

To temporarily bypass this dilemma, we shall use one dictionary only and thus there will be a final say on whether or not a certain combination of symbols is a word. When taking a group of words together, we next form what is referred to as a sentence. Not all combinations of letters form valid words, and similarly not all combinations of words form valid sentences. Again the determination is based on certain rules, which can be found in various books on the subject.

As you can tell there are many rules. That may be why English is such a difficult language. The rules don't end here, as now sentences will be put together to form paragraphs. Not all combinations of sentences will form meaningful or valid paragraphs and once more we need to follow guidelines, which have been set up in defining the language. But assuming we have some valid paragraphs, these put together will make up what is referred to as a chapter. Obviously there are more rules in determining this composition, just as before.

Now taking a group of related and meaningful chapters, the combination will result in a novel or work of nonfiction. We now have what is referred to as a book and I shouldn't have to remind you of the necessity of following certain rules in order to achieve a meaningful book. The last grouping will give us our library, that is, putting a set of books together gives us this structure. Assuming all our books pass the test of "validity," at this point we have no special rules as to what can go into our library.

Some might say that I missed a few groupings such as putting words together to form a phrase. What about bunching three novels together for a trilogy or a set of works together to get a volume? Why not put all the psychology books in one department and young adult fiction in another? You would have a very valid point but I am just trying to outline the main tenets of a language. As I said earlier, all languages will follow a similar set of rules, whether they are a foreign language or a computer language.

Just because there are rules for forming valid words and sentences and the like doesn't mean that everyone conforms to them. I have worked with many people who make up words. I'm sure you have too. These individuals use so-called "words" and "sentences" as though they were as common as all those that are valid. This does make for frustration and confusion and lack of understanding. However, it does gives me plenty of material for my books. If you've read any of them, you're aware that I have a great deal of fun making up words. Someone has to do it. My 2005 book, ***for seeing eye dogs only*** and its two sequels deal with missing intelligence as well as oxymorons, acronyms, pleonasms, words and near words. There's another combination that I just heard about recently, but it's not included here because I can't spell it.

Corporate America has its own set of words and phrases, but good luck finding documentation anywhere. This makes it extremely difficult to figure out exactly what they mean. If you are part of the business world as I had been for over twenty-five years, mostly as a consultant, you may find it difficult in determining what people are talking about. If you are outside the environment and you try to understand what is being said, it's even worse. Perhaps that's why big business has so many problems.

If a language has no rules, you will never be able to use it or understand it. Too many rules mean that there will be rebellion and once again you may as well have no rules. Obviously there have to be some rules but there is a limit. You can't have too few precepts nor too many. That middle ground will result in a successful language that meets the needs of a group of people. This applies to everyday communication as well as the languages of computers.

2. Our programming language

English works with certain rules, and so do programming languages. It will not matter which one you're talking about, as they are all similar. Since this work will not be affiliated with any specific one, we'll deal with a hypothetical computer language, which we'll call *P language*. It will have very specific rules, which we shall introduce from time to time. Learning it should give you a good grasp of what any other computer language involves. Since computer systems encompass a vast area of knowledge, we shall only cover a small subset, namely programming.

Just as there are basic elements to English, P language has those same constituents. Our language will use the letters of the alphabet *a* through *z* and other special characters. We will not need capital letters but if ever someone uses one by mistake or otherwise, that will not be a problem. Our system will simply assume it's the same letter as lower case. The letters of the alphabet will be used together to form words, just as in English. There will be three resulting uses of these words. The first will stand for a variable – which shall always be represented by an underlined word. Once we get into a program, you'll see the underline used. A variable will represent a value for a field. We could use the field or variable

balance

to represent a bank balance for someone's checking account or the variable

interest-rate

could be the rate the bank pays on this checking account. Note that both these fields will change in value and that's why they're called variables.

The next use for a word will be for something very significant in our computer programs. These represent concepts – we'll get into them shortly – as well as verbs dictating specific action to be taken. Referred to as keywords, each will be in bold print. Thus

print

might be used to get a report printed on a piece of paper. Whenever it is used, it will always accomplish the same thing, that is, produce output for a report. For that reason, these words are defined to our system and cannot be used for ordinary variables. They are keywords or reserved words. Usually a system has a list of all these words. Our system will be no different and a list can be found at the back of the book. We will define and describe these verbs as we use them.

The third use will be for a label of a paragraph, which we'll get to later. We'll also use operators – usually a single character – to do a few things, such as addition or multiplication. To add the variable

deposit

to

balance

we could write

deposit plus *balance*

but instead we shall say

deposit + *balance*.

As you can see, our operator is the plus sign. There will be operators for subtraction and division as well as logical operators, which are used to make decisions in programs, when we need them. We will get into these later.

Hence, we have variables, keywords, labels and operators. Variables can use any letters of the alphabet, numbers as well as the hyphen. No other symbol will be allowed. Each variable must begin with a letter and cannot start with a hyphen or number. The following are all valid:

initial-balance
deposit
jxqrtk
x
x-1-y-2

Each of the following are invalid:

bank balance – it has a space or blank between the end of one word and the start of the other and that is not allowed

3rd withdrawal – the first position is a number, which is not allowed

x – 1 – the spaces around the hyphen are not acceptable

in&out – the *&* symbol is not allowed in variables

As far as the size of the field, there will be no limit; but some considerations are in order. If you use *x* for a variable, it will be valid, but it might be difficult to understand what it represents. If it is to stand for monthly maintenance fee, why not use *monthly-fee*? For a due date you could use *z* but *due-date* will be more appropriate. It will be more meaningful. Thus a rule to use will be to make the field name long enough to have significance but don't forget you have to key it in, so don't make it too long either.

As far as keywords and operators go, the former by their very makeup should be easy to figure out regarding what they do. Usually operators will be a single character. If there is any doubt as to the meaning of either of these, refer to the index at the back of the book for a list and descriptions of keywords and operators.

Putting together variables, keywords, labels and operators will result in a phrase or sentence, not unlike the English language. In our case though this will become a line of our program. Note that there will be rules to follow for each line and so far there has been a hint of some of these constraints. We shall get into more specifics later. Taking a group of valid lines of code and assuming some rules are followed, the result will be a section or paragraph of our program, just as we had for the English language. With more rules being met, a certain group of paragraphs or sections put together will result in a computer program, which parallels our chapter in English.

Finally putting a group of programs together with further considerations will result in a user application. This is very similar to our novel or work of non-fiction in English. We could proceed further by grouping a few applications together to give us a computer system. This we saw as our library in English. Our concern in this work is programming so we will concentrate on that aspect and only mention applications and systems on occasion. You can see that P language and all it encompasses is very similar to what is involved with English. There are many similarities.

3. File makeup

Before proceeding with a simple computer program, let us look at how data is organized. All information is stored in files or databases, which strictly speaking are one and the same. A file consists of various elements or records. Thus a personnel file will have records that match individuals. Each record consists of fields or variables. Our personnel file might have records that include some identification number such as a social security number or the like, name, address, city, state, zip code, telephone and date of birth. There may be other fields as well.

Each field is a variable, which has a value, and each individual field has some kind of limit. The identification number might be limited to nine numeric digits and nothing else. It cannot be all zeros or all nines and there could be further restrictions. The name will be limited to letters of the alphabet – upper and lower case – the period, apostrophe and hyphen. I don't know many people who have a name with $, %, a number or @ in it, so I think our restriction is valid. There is allowance made for hyphenated names to accommodate women who marry and want to somehow keep their maiden name as well as an Irish name like O'Brien. Granted, there are taxi drivers in New York City who have the letter *O* with a slash through it in their name, but we won't concern ourselves with that possibility.

Other fields will have different restrictions. Zip code can be one of a few formats, such as five digits, nine digits or alternating digits and letters to accommodate our neighbors north of the border. Dates have to be in a specific format, mostly all numeric but all spaces could also be acceptable, as could an entry of all zeroes. This would accommodate a date to be entered later. Our language will require all dates to be in yyyymmdd format, that is, four digits for the year and two each for the month and day. If the date is neither zero nor spaces, MM, DD and YYYY have to be such that their combination is a valid one. MM = 02 with DD = 30 would be unacceptable since February 30th is not a valid date. Later we will develop a date check to handle this.

Other fields will have restrictions as well. The state has to be a valid two-character combination, which represents one of the fifty states. City can be no more than fifteen characters and these can only be letters of the alphabet, the hyphen, the period and a space. Amount fields will always be numeric and some can be negative, such as a bank balance. Thus some amount fields need to be able to be positive or negative. This is handled by including a sign in the field. Amount fields have decimals in them, such as current balance, so that will must be taken care of as well. There will be no need to put the decimal point into any file just as we don't need to include a dollar sign for a withdrawal or deposit. Since we are talking about money, the $ is assumed.

Having delved into the structure of a file, you can probably see that the makeup is not unlike the book we talked about in the English language. Each has basic elements that make up words or fields. These pieces in turn then get grouped together to form sentences or records. English then combines the sentences to get a book while the combination of our data records makes a file. In each case there are rules that need to be followed. If we fail to follow the rules for either, there will be problems.

The file that we want to consider is a file for checking at the bank. For now it will consist of just a few fields, account number, last name, first name, middle initial, street address, city, state, zip code and balance. Using someone's social security number – because of identity theft – is not a good idea. In some cases, the computer will generate an account number – and even let the customer know what it is. In our system, the account number will be a nine-digit field greater than nine.

Both the first and last names must consist of letters of the alphabet, the space, apostrophe, period and hyphen only. This accommodates Billy Bob Thornton, Tom O'Brien, Jill St. John and Olivia Newton-John. The first name is limited to fifteen characters while the last name is restricted to eighteen. That should be enough characters. The middle initial must be *A* through *Z*, but it can also be left blank. The street address is limited to twenty-five characters and has the same restrictions as the name, except numbers are also allowed as well as the comma. If you live at 10 ½ Main Street, good luck. City must be no more than fifteen characters and these must consist only of letters of the alphabet, the period, space and hyphen.

The state must be exactly two characters and it must be the valid abbreviation for one of the fifty. The zip code must be a five digit numeric field. The balance will be a signed numeric field having eight digits, six to the left of the decimal point and two to the right. If you have a balance of over $999,999, it shouldn't be in a checking account. In fact this bank may even be more restrictive and caring about the customer – that could happen – as large balances might result in letters being sent out notifying customers that they may want to consider a certificate of deposit or the idea of buying stock.

Our file is the account file and if I want to read it in a program, I will specify the variable

acctfile

that represents a file which the program can read. How this is done will be shown when we get to the program. For now we need to worry about the fields that make up the file. We have to spell out the names of the fields, their sizes, where they are in the record and what type each field is. To save space one field will follow the other so we'll define a structure, which will refer to the file, the record and each field.

We'll define a file and its composition so that we know the makeup of a typical record. That way, we'll know where each field should be. We certainly don't want the first record to have the account number at the beginning followed by the last name and then the second record have the first name directly after the account number. That scenario will make it impossible to process the file. In our account number file, the account number will start in position 1 of the record and end in position 9, last name will start in position 10 and end in position 27, first name will begin in position 28 and end in position 42 and so forth until we get to balance, which ends in position 99. This will be the case for each record on the file and it means we can find the data we want where it should be.

We could have put commas as separators between the fields and accomplished the same result but what happens when one of the fields has a comma in it? That could mess us up so our method will be better. We start by defining a file and its structure. The

account number file consists of nine fields. We must then thoroughly describe each field. This gives us some keywords. The first is

define

and the others are

structure,
integer,
decimal,
signed

and

character.

The actual program code to describe the account file record and its makeup is as follows:

```
define acctfile record account-record structure
        account-number integer(9)
        last-name character(18)
        first-name character(15)
        middle-initial character
        street-address character(25)
        city character(15)
        state character(2)
        zip-code integer(5)
        balance signed decimal(6.2)
```

Note that the ten lines above are not a program, which we'll get to in the next chapter. Let us begin with the first line,

define file acctfile **record** account-record **structure**.

The keyword

define

spells out to the program the name of the file – indicated by what follows the keyword

file

and what fields make up each record. That's what the keyword

record

is for. The field

account-record

is a variable, as are the nine fields in the record that follow. The record is related to these fields by the keyword

structure

which says that the variable

account-record

consists of nine fields. The end of the record is indicated by the next occurrence of the keyword **define**,

or some keyword, such as

read.

The line

account-number **integer(**9)

has the variable

account-number,

which is the first field in our record or structure. Because of the way the structure is defined, this means that the field

account-number

starts in the very first position of the record. The keyword

integer(9)

spells out that the field is a number consisting of 9 digits. As you may have guessed

integer

is another keyword. Any number that is an integer is a whole number, which can be 0.

The next line,

last-name **character**(18)

is quite similar except this field is not numeric but rather consists of letters of the alphabet. The keyword

character

is all encompassing and just about any symbol will be allowed in using it, even a number – even though, as I write this, people don't have numbers as part of their name. Seinfeld fans, that show is fantasy. Later, we'll see that numbers in the last name, first name or middle initial aren't allowed, even though this keyword will include numbers and special characters. Note that this field contains 18 characters maximum. If the last name happened to be

Smith,

last-name

would consist of the letters "Smith ", that is, those five letters followed by 13 spaces.

The variable

first-name

is similar to

last-name

except that it only has 15 characters.

middle-initial

is a single character, so we could have spelled this out as

character(1)

but

character

represents a single position as well. The next four fields mirror the others above them, as they are either

character

or

integer.

Note that both variables

account-number

and

zip-code

could have been defined using the keyword

character

rather than

integer

since each number is included in the character set. The last line

balance **signed decimal**(6.2)

introduces two new keywords,

signed

and

decimal.

Since the account balance could be negative at times and it does involve cents as well as dollars, we need to spell that out. The variable

signed

allows for negative as well as positive numbers, while

decimal(6.2)

indicates that the field has 6 digits to the left of the decimal point and 2 to the right. If the balance happened to be $16.20, it would be on the file as

00001620,

and because the field has a sign, the program knows that this is the positive value of

16.20.

It knows exactly where the decimal point is even though it is not on the file.

This structure will be used in the first program that we consider and we'll be using other structures as we need them. This will enable us to read a file and access specific fields in each record. The keyword

structure

merely says we have a piece of a file, namely a record, and this record itself consists of different things. These are nothing more than the fields that make up the record. We can access the entire record or individual elements of it.

4. A report program

You can do a great deal with a computer program but in general all programs do the same thing. They read data and produce output, either another file or a listing on paper or on a screen. In the process, sometimes files are updated. The data may be obtained from the system somehow, from screen input or from another file. Despite this simple breakdown, the process could get quite complicated. A program could involve reading a dozen files to get different kinds of data and then produce a few files. A data entry program might involve a screen to allow input, which could be quite complicated. It really depends on the system and what kind of computer you have. Fortunately our system will not be complex and so you may be a little upset to see more complexity when you get to a different system.

Here are a few examples to illustrate this fact. It is discovered that someone messed up the account file for all the begin dates. Instead of 2000 or 2001 for the year, each record has 0000 or 0001 for this value. Someone writes a program to correct this situation. The input to this program is the account file and the result will be either a new one or an updated account file. This all depends on what type of file it is. In either case the two files will be different, with the new one having the correct begin dates. Thus the program will read a file and create an output file. There could be another output file, namely a report to list the records that were changed.

The account file needs to be updated with new accounts from time to time so there is a program to allow input into the file. Once again we have an input file in the data being keyed and probably two output files, the updated account file as well as some kind of report. Even though the report file is not completely necessary, it is probably a very good idea to show the addition of the new accounts.

Our first program will read the account number file and produce a listing of the fields on it. Specifically, we will read a file and produce output in the form of a report, but just one record will be listed. That’s very restrictive, but we’ll get into reading the entire file later.

```
program-name: acctprint
define acctfile record account-record structure
        account-number integer(9)
        last-name character(18)
        first-name character(15)
        middle-initial character
        street-address character(25)
        city character(15)
        state character(2)
        zip-code integer(5)
        balance signed decimal(6.2)
read acctfile into account-record
print account-number
print last-name
print first-name
print middle-initial
print street-address
print city
print state
print zip-code
print balance
end
```

The output will look like the following:

```
391023123
smith
chris
t
396 main street
buffalo
ny
14225
00001620
```

Obviously some explanations are in order, so let us start with

program-name: acctprint.

As you could guess

program-name

is a keyword that we use to indicate the name of our program. We will be writing many programs so we need to distinguish one from another. We do this with that keyword. The name we choose here for our program is a variable,

acctprint.

When we write another program we will have some other name and this is needed to keep the programs separate. Note that we choose this name because we are listing account information. It's a good idea to make your choice meaningful, as that will help you later when you have so many different programs.

The next few lines should be familiar as they describe the account record and all the fields that make it up. Through the

structure

keyword we can reference any of our nine fields in the record. We need to do this in order to print them on a report. Note that the end of the structure will be determined by the keyword

read.

We then have two more keywords,

read

and

into

in the line

read acctfile **into** account-record.

The variable following

read

is our account file which has the name

acctfile,

another variable. This line actually does two things. It opens the file and then reads it into the account record layout, making all the fields available to us so we can put them on a report. Using just

read acctfile

would accomplish the same result because of the way the file is defined. The keyword

record

ties the field,

account-record

to the account number file so that any read of that file will result in the data being moved to the field,

account-record.

In the world of computers, there are numerous ways of doing things, which can be good and bad. This I pointed out in an earlier chapter when I talked about systems and rules.

The next nine statements are all print statements using the keyword

print.

Hence the first one will print out the account number, which happens to be

391023123

in this case. The remaining eight print lines will then list the remaining eight fields in the record, as shown on the listing above. Note that the last field is the account balance and it is

1620,

which indicates an amount of

$16.20.

The very last keyword is

end,

which will close the account file and end the program. That is all there is to the program with the main activity consisting of a read, nine print lines and an end. This program simply opens and reads the file into an appropriate layout, prints out the nine fields on the first record, closes the file and ends – truly exciting stuff.

There are a few concerns. For one, what about the rest of the records in the file? Second, it might be nice to have some labels for the fields on the report so we know what each is and it may be better for the environment to print the fields across the page rather than down it. While we are at it, what about a title for the report? Why doesn't the account balance print with a decimal point and without those leading zeroes? Lastly, why do the names begin in lower case letters rather than upper case and what would happen if the account file had no records or didn't exist at all? These are all valid questions, which need resolving.

Let us begin with the question about upper case in the names. The reason they are lower case is because someone entered them that way. We can resolve that in one of two ways by either reminding the data entry people to appropriately use capital letters in these situations or we could change our data entry program to make sure that these first characters are upper case on the file no matter what is entered. Needless to say it is important to enter correct data otherwise our file won't have much validity. You've heard the expression, "If you put junk in, that's what will eventually come out."

Before continuing, let me clear up one point relative to upper and lower case. You may remember that I said we needed only lower case before. And yet how do we account for the first letter of the name without capital letters? The restriction on using only lower case applies only to our computer program, not to the data itself. We can certainly get along well with that limitation. However, we probably need to allow both upper and lower case in the data that will be on our file.

We could print a label before the account number by the statement:

print "account number: " account-number

and the output would then be

account number: 391023123

which is an improvement over what we had before. The double quote enables us to print literals or strings of values that can be helpful in reports. It may be hard to see, but note that there is a space after the colon, which keeps the label and the actual account number from running together.

This won't put all the fields on one line but we could do it by two print statements. The first would print all the field labels and the second would print the actual field values. Since a line on our report has space for about 130 characters and our record layout is 99 characters, we should have enough room on one line to print all the fields and have room for spaces between the fields as separators. This we can do by the following print lines:

```
print "account #   last name          first name       mi  street address            "
          "city            state zip    balance"
print account-number "   " last-name "   " first-name "   " middle-initial "   " street-address
          "   " city "   " state "   " zip-code "   " (balance, mask($$$$,$$9.99))
```

The output would now be:

account # last name first name mi street address city st zip
balance
391023123 Smith Chris T 396 Main Street Buffalo NY 14225
$16.20

Note that the line on this page does not have 132 characters so what you see above is not exactly what you would see on the actual report. The word *balance* and the value of it would all be on the same line with the other data and there would be more spacing between the fields. Also notice on this page that the headings for each field with the appropriate value don't exactly line up. This is due to limitations in the word processing software that I am using, which I can't do much about. Nobody said computers are perfect. If the page you are reading does not have this discrepancy, it means that the publisher took care of the problem. In any case, I think you get the idea.

There would be three spaces between the fields and more depending on the values, specifically the name fields, which allow for more than the number of characters that each name actually has. Thus there will be exactly four spaces between the # sign and the label *last* and exactly three spaces between the last digit of the account number and the first letter of the name, *Smith*. Note that we have our upper case designation for the names, which means someone entered them correctly on the file.

Though our first print statement takes up two lines in our program, it will all print on one line of our report. The same is true of the second print statement. The reason that we have two lines each is because they physically don't fit on one line. If we had put the keyword

print

before the literal *city*, then the labels of the fields would have printed on two lines which is not what we want. On the second statement which print the values of all the fields, we have a specific literal,

"   "

printing after each field to separate the fields. This string consists of exactly three spaces. I mentioned the keywords,

character,
integer,
signed

and

decimal,

before, so you have a good idea what they stand for. In summery, the first can be just about anything, any letter of the alphabet, a number or special character.

integer

is any single digit, positive number, including zero.

decimal

allows us to talk about the numbers between whole numbers, such as *3.23*, and

signed

expands our system so that we have negative as well as positive numbers.

The line handling our balance needs some explanation and it ends in

(balance, **mask**($$$$,$$9.99)).

This will give us the account balance in a way that we'd like to view it. The keyword

mask

is used to reformat the balance so that it will have the appropriate dollar sign, commas if the amount is a thousand dollars or more as well as the decimal point in all cases. The dollar sign used the way it is here allows it to float to just left of the significant digit in the amount. Recall that the record has

00001620

for the balance, so with the mask, the leading zeroes on the left are not printed and the dollar sign winds up just to the left of 16.20. The

9

in the mask forces the number to print, even if it is a zero. Hence a balance of a quarter would print as

$0.25

using this mask. The decimal amount will always print with this mask. Note also that we need two right parentheses to end the statement in order to balance those two parentheses on the left and the mask is enclosed within one set of parentheses. The outer parentheses are needed to assure that the mask goes with the variable,

balance.

Using this same mask, an amount of three thousand dollars and ten cents would print as $3,000.10. We could choose to not print the dollar sign or leave out the comma and this we could do with a change to the mask, using

mask(999999.99).

Our values would then be

000016.20

and

003000.10

respectively. To suppress the leading zeroes we could change it to

mask(zzzzzz.99)

and now we would get

16.20

and

3000.10

for our formatted amounts. The character *z* here just suppresses the leading zeroes.

To create a main title for our report we could simply add another print statement. It might be

print “ Account balance report”

and note the few spaces to the left of the word *Account* we used to guarantee that the heading is centered on the page. We shall talk more about headings in another discussion but now we must accommodate those other records on the file that we didn’t list on the report and the possibility of an empty file or none at all.

We do this by introducing more keywords,

status,

if,

go to

and

end-if.

We do have three possibilities here, that is we could have a normal account file or an empty one or none at all so the keyword

status

will enable us to see which of the three comes into play in our program. The last three keywords will give us the ability to make decisions and even branch if necessary, which we will need to do in many cases. The main logic of our program (the code after the structure and its definition) now is:

```
define acct-status status acctfile
print “          Account Balance Report”
print “account #   last name        first name      mi  street address          ”
      “city          state zip   balance”
account-number = 9
read-file: readnext acctfile
      if acct-status = 0
            print account-number “   ” last-name “   ” first-name “   ” middle-initial
                  “ ” street-address “ ” city “ ” state “ ” zip-code “ ”
                  (balance, mask($$$$,$$9.99))
            go to read-file
      end-if
      if acct-status not = 9
            print “the account file is not integral – program ending”
      end-if
end-program: end
```

The first line

define acct-status **status** acctfile

defines a two position numeric field for

acct-status

which refers only to the file

acctfile.

The status – which we don't define – can be anything from 0 to 99. This is done by the keyword

status,

which is always a two-digit number, or

integer(2),

that we will use to verify that any processing of a file has no problems, whether it is a read, write or delete. Here

acct-status

will be used to see if we have a successful read. A value of 0 will indicate that the read was error free. In fact we shall see later that other accesses to the file such as a write will also result in 0, provided they are successful. If we read the file and there are no more records left, the record status will be 9, indicating we have reached the end of the file. Any other value that results means that the file has a problem and we can't continue in our program.

Let's look at the lines

```
account-number = 9
read-file: readnext acctfile.
```

The first is an assign statement, where the variable on the left is given the value 9. The smallest account number is 10, so the

readnext

verb will try to read a record with an account number of 9, but since it can't find it, it will read the next record. In the second line, the first part

read-file

is a label, which is used since we need to get back here over and over. We could have called it "xyz" but the name we assigned is much more meaningful. Labels are followed by a colon.

program-name

was also followed by a colon, but since it is a keyword, it is in bold.

The next six lines work hand in hand.

```
if acct-status = 0
        print account-number "   " last-name "   " first-name "   " middle-initial
                " " street-address " " city " " state " " zip-code " "
                (balance, mask($$$$,$$9.99))
        go to read-file
end-if
```

The keyword

if

gives us the ability to do something depending on a comparison. In this case we are looking at the field

acct-status

to see if it has a value of 0. This means that the read of the file was successful. If it is, the new line or two indicates what action is to be taken. In this case we have the fields we need to print a single line of our report. We print that line and then proceed to do another read. This is accomplished because of the next keyword,

go to,

which allows us to branch to the label

read-file.

We can now try another read and proceed as before. The keyword

end-if

is used to indicate that our if statement is complete.

The next three lines

```
if acct-status not = 9
        print "the account file is not integral – program ending"
end-if
```

interrogate or check the value of the field

acct-status

and if it is not equal to 9, there is a problem with the read of the file. In this case we cannot proceed so we print out an error message. The

end-if

again means that this particular if statement is done. You might say that we should end the program and that's exactly what will happen since that's the last line of the program. If the

acct-status

is 9, indicating the end of the file, we will wind up in the same place – exactly what we want. Note that if the

acct-status

is 0, we won't get to this point since we will have branched back to the label

read-file.

You may be questioning the use of

$$$$,$$9.99

rather than

$$$,$$9.99

for the edited balance. Remember that we need one character for the *$* and then one each for the six digits to the left of the decimal point. That is why we need the six dollar signs and one *9* or seven places in all. If the balance were $100,000 and we used

$$$,$$9.99

as the mask, the result would be printed as

$0.00

since we have only provided for six positions, but we need seven. As a result, the leftmost digit would be truncated, which is not what we want. The computer will do exactly what we tell it to do. It can't correct our omissions, such as this.

The last line of our program

end-program: **end**

simply ends the program and closes the file. We saw it in the earlier version of this program. As you will agree, these modifications are a huge improvement over what we had. We're not done yet.

5. File access

If you work with different computers you will hear about flat files, sequential files, indexed or keyed files and databases. That's only the beginning. The first designation is not used to represent a file that was run over by a steamroller but rather a simple file that we can read one record after the other and can't update. There are no keys in the records of the file so we can't read a specific record without reading every other record, at least until we get to that record. This is also what is referred to as a sequential file. These types of files can be used quite successfully to back up a file and restore it and either can be done quickly. An equivalent music device is the cassette or eight-track, each of which results in listening to every song in order or fast forwarding to get to the song you wish to hear. I'm not sure where the term, *flat file* originated, but why do we need the designation when the term *sequential file* suffices?

The next type of file is an indexed file or keyed file, which has some sort of key in it. This enables us to get to a specific record without reading every other record in the file. This could save us some time since we could obtain the record we want quite quickly, but we have to know the key to the record or at least part of that key or some other significant field. If the key was the account number and we didn't know it but we knew the customer's name, the computer program could be intelligent enough to give us a list of accounts to chose from and one of those could be the one we wanted. Many systems give you this option. An equivalent music device is the record or CD since either can get us to a specific song without much effort, unlike the cassette or obsolete eight-track.

If you have a keyed file, the keys are usually unique, that is, you won't have two records with the same key. Nonetheless you can have a file that is an indexed file with duplicate keys. There is a reason for this, which I won't get into. Just be forewarned. There are all kinds of indexed files and the differences are due to the company that developed them or the time when they came out. If you know one index file method you can adapt to any other.

The last designation is a database, and as I mentioned earlier every file is a database as each has *data* that is a *bas*e for our system. Some will argue that a database needs to have a key and this equates to an indexed file, but certainly a sequential file is a database – with limitations. Thus, every database is a file. The distinction between files and databases is a very fine point, which I won't belabor.

If you work with other systems, you will note that the program using a file may have to open it, read it and finally close it. The language that uses this file may actually do a close of the file as the program ends just in case you somehow forgot to do it. This suggests to me that the close that seems to be required is not really necessary. In our sample report program earlier we neither had to open nor close the file because our system is quite intelligent, which is what all systems should be.

For our system, all the files will be indexed files. They will all have unique keys and we can access records in the files by keys as well as read those files in a sequential manner. That is exactly what we did in our very first program to list the fields on the Account balance report. We will get into processing data by specific keys later. The file we used in the previous chapter was also processed sequentially. In our system, the field

account number,
will always be generated by the system. If our report had fifty accounts, they would all be in ascending order with the lowest key first and the highest last. Recalling the restriction on account number being greater than 9, there is a very good chance that the first record would have an account number of 10, followed by 11 and 12. However there could be gaps in the numbers, as we shall see later.

Some computer systems will lock you out of a file if someone else is updating it. Thus if someone was updating a record in our account file, we may not be able to read any record in the file. Our system will be a little more permissive, having been designed by liberals. If someone else is updating the record with account number 395123867, we won't be able to update that specific record but we can read it and we can read or update any other record in the file. If two people are updating the file at the same time, most likely they won't be on the same record but if they just happen to be, we need to take some precautions.

If two people want to update the record with account number 395123867 at the same time, one of the two people will get to it first. Let us say that Pat is that person and he changes the zip code from 14225 to 14229, but he hasn't done the actual updating just yet. Just before Pat completes the update Chris accesses the same record and the zip code still has the value 14225. She changes the middle initial from L to P and Pat does his update, resulting in the new zip code in the record. But then Chris does her update and the middle initial now is P but the zip code has been returned to the value of 14225, not what Pat had intended. The changed value has been overlayed. We cannot allow this to happen and I will get to how this is handled when we work on an update program. I think you can see that locking the record temporarily should work and not locking the entire file means that both can update different records at the same time. That would be the way to design the system.

Designing the files is done by a DBA or data base analyst. He or she does this with input from other people for efficiency. After all you don't want a file designed that requires long waits by the users in getting at the data. You also don't need redundant data, as a field that occurs twice in the file just uses precious space. You also don't want to keep changing the file definition month after month. This means time needs to be spent on analysis and design. In our account file our key may actually be larger than we need but that is something that needs research. I recall a system that I worked on that had three positions for a transaction code when two might have been sufficient since there weren't any transaction codes bigger than 99.

That whole consideration of trying to save two digits for dates by using only two positions for the year instead of four is what caused the Y2K fiasco. I won't get into that but you can see where time spent planning can save a great deal of time later. There is much to be considered and if you're working on a project where all the ideas and design of the system are not firmly in place, it will be impossible to come up with a database design that will suit everyone and keep management happy. The design of the files will have to wait.

These are just some problems involved in information technology systems and you will run into them no matter where you work. The better the system is thought out,

the more pleasurable will it be to work there. By the same token, there may not be that much work for you because of that. The places that have plenty of opportunity for work will probably be the corporations that you would not rather set your foot into. What a dilemma.

6. Program abends

While studying computers at the State University of New York at Binghamton, one of my classmates had a cat name Abend. Some other people I met had a cat name Cat and a dog named Dog, which probably didn't take much thought. I thought Abend was an appropriate name since we were studying computers. The word *abend* is a contraction of the phrase *abnormal end*, which many programs and systems do and with which not many people are happy. It means working overtime and on the weekend. If you read my book, ***Tick Tock, Don't Stop: A Manual For Workaholics***, you probably know that working more than thirty five hours a week doesn't thrill me too much. My second book on work, ***This Page Intentionally Left Blank – Just Like The Paychecks Of The Workers***, advocates a thirty-hour workweek, which I think is a better idea. You may have heard of Timothy Ferriss's, ***The 4-Hour Workweek***, but that may be a bit drastic and cause a few headaches. I doubt that management would approve.

In information technology there are quite a few ways for abends to occur. A program could encounter bad data when it is looking for a numeric field and instead finds letters of the alphabet. The result is the program terminates. A system could run into a space problem and the result is an abend. There could be an I/O problem in that a program is reading a file on magnetic tape when the read fails. The cause may be something as simple as the fact that the tape drive missed its normal maintenance cleaning, but it could be something else.

There could be a disk crash or you could run into bad sectors on the disk and the results could be really annoying. I had the hard drive on my personal computer replaced a few summers ago and it wasn't much fun. The word *crash* seems to be another way of saying that we had an abend because they are one and the same. I was editing a page of this book on my word processor when I tried to save what I had added and couldn't. The only way out was shutting down and restarting, which resulted in my recent changes being lost. And I thought computers were supposed to make our life easier.

Each of these possible scenarios has to be controlled. If not, there is no sense in having a system because reliability is compromised. You might be spending more time with your computer than if you had a manual system. Obviously there will be difficulties from time to time and you will have to put up with the problems, but you need to do everything possible to limit these troubles.

To avoid space problems you have to maintain files, eliminating those that are out of date or redundant. It may be as simple as backing up these questionable files to a tape so that if they are needed, they can be retrieved. This leads to valuable space being saved. Another kind of maintenance has to do with backing up a file and then redefining it and restoring it from the backup. What this does is eliminate fragmentation, which happens to files when they happen to have gaps in the data or be on multiple disk packs. One of the maintenance tasks on PCs is checking for fragmentation on the disk from time to time and taking action if necessary. Another way of helping alleviate the space problem is eliminating duplicate data on a file, which I will get into later. There's much that can be done and it will depend on your environment.

Avoiding system crashes is almost impossible but if you have some kind of recovery technique that can minimize the damage, you will be one step ahead of the game. You won't be able to tell how reliable a specific disk from some manufacturer is until you use it. However, talking to other corporations can give you some insight as to whom to avoid and who may have a worthwhile product for you to use. You'll have to do a great deal of homework but it will eventually pay off.

Power failures could cause nightmares so you need some way to handle them. The solution may be as simple as backup systems on temporary generators so you won't even feel the effects. Unfortunately it may not be that simple. You could be blessed with so few outages that it won't even be a concern. Consider yourself fortunate.

You won't be able to live without backup of your files and systems. If you have programs that update files, backups at the appropriate time will save you from a great many headaches. You may need to restore files to some point in time but you won't be able to do that without the right backup. If you use the one from the weekend, you could lose a great deal of data and time even though some files have been restored. Once again planning your system should reduce problems to a minimum.

Despite all precautions, even the best intentions can result in major screw-ups. In the early 1980s, I worked full time for a computer company in the software department. My assignment was that of a consultant without commensurate pay. I quit the company after about a year and a half when I saw that there seemed to be no future in this specific business. While I was in their employ, I was asked to create a purchase order system and I was given two options: modify an existing system that the company had or write one from scratch. I was specifically directed to determine my approach with some analysis, so I set forth and discovered that using what was there would have required more time and effort. Nonetheless, my boss insisted that I write the purchase order system using the program in the software package at our office – that wasn't a good idea.

I should have wrote the system my way and not told a soul about it but instead I foolishly did as told. The result was that my boss complained when it took longer than he wanted me to spend. When I was done though, the effort was top-notch and the clients were pleased as punch – whatever that means. They used what I produced and couldn't have been happier with my creation.

Unfortunately, Murphy showed up one day. You've all heard of Murphy's Law, so you know what I'm talking about. What happened was that the client ran into a problem one day and had to resort to the backup of their system. Their practice had been to do a daily backup of the system in case of any problems. When they tried to restore the system from the backup, it turned out that it wasn't worth spit. They went back a few backups but even those were as worthless as Tom Delay being in charge of the ethics committee. Going back like they were doing meant that they would have lost a few days activities, but at least they had a system.

The problem was that the tape drive backup wasn't working and it hadn't been for some time, even though it appeared to be fine. I'm sure you've seen numerous examples of technological processes that seemed to be working when in actuality nothing was happening. That was just what was taking place with the purchase order system daily backup and there was no suitable backup. Fortunately, the office where I worked – at the

time – had the purchase order system still on their computer. Well, *had* is the operative word because one of my co-workers deleted it there. He didn't do a backup first but simply eradicated it. As you can guess, that wasn't very intelligent.

By this time I had left the company but was contacted about the fiasco at home. I mentioned that on the desk where I spent my days at the office was a tape with the purchase order system. Restoring it meant that the client would have to start data entry from the beginning, but at least they had their system back. You probably guessed the ending of this tale, but if not, I need only mention Murphy. Sure enough, someone had used the tape by writing over it and the system was lost forever. I didn't return to the company to redo the system and I am not sure of the relationship between the company where I had worked and the client. I do know that within a few months this computer company bit the dust.

Returning to our discussion from my digression, I/O problems may be reduced by consistent hardware maintenance but bad data will require considerably more effort. You should never have a program that abends because of bad data caused by a field generated by the system. Any program that accepts input to be used as fields in a file should guarantee that this field will never result in an abend due to bad data. If the field is numeric, the program should never accept any other type of data into this field nor should it ever write out this field to the file if it is not numeric. This may take a bit more programming but it will result in evenings and weekends with virtually no calls from the computer room. That will make your time at home more enjoyable.

Problems will occur no matter what you do and no matter how careful you are designing systems. The steps you take will differ from system to system but you need to minimize the effects of crashes and software failures. As we get into further sample programs, I will offer some suggestions to help reduce the frustration without leaving your place of employment or jumping off a twenty-story building.

7. The online account display

Looking at the account number file report but you may ask why we didn't save paper and display the fields on the screen. To save the destruction of trees for paper for our reports, our system will create the report but not actually print it to paper. Instead it will display data on the screen. You can get a full report printed but it will cost you. The intent is to make the cost so prohibitive that people will think twice about getting it on paper. For a lesser amount you can have just the page you need printed. This will cause people to think about the reports they request and be more concerned for the environment. At the end of the chapter, I will talk briefly about going paperless.

Obviously we need to look at data on files but we can do that without looking at a piece of paper. I never want to read a novel online but data in the office is an entirely different story. How to get the report on the screen will depend on the system you have so I won't get into the details. We can peek at records on the account file if we have some kind of inquiry program online. Since our file is indexed, we can look at a specific record if we know the account number.

Before getting into the actual program, let me talk about program specifications. These are nothing more a detailed summary of what the program is supposed to do. They can be very long, encompassing not only what is to take place but also how that should be done. On the other hand specifications can be brief, to the point and it is up to the programmer to do the rest. There's a joke in the computing business about specs written on the back of a napkin and I'm sure that some have used that medium.

Our specifications can be written as

Write a program to display account file fields based on the account number.

It could also be

Transfer the original account file report to the screen but limit it to one record at a time depending on the account number input.

Vague as these specs may be, in either case you get the idea of what needs to be done and how it is to be done is up to the programmer as well as where to place the fields on the screen. There's quite a bit of freedom in getting the data to appear but you can understand that we don't want a cluttered screen and it should be user friendly. After all, we don't want to do the programming and later have some person say they are not happy with the way it looks. Actually, our goal is to have the user be so thrilled that she holds a party to celebrate the occasion.

In order to get data on the screen we need the keyword,

screen.

The computer monitor in most cases has 24 rows and 80 columns to display characters. Reducing the size of the characters can increase the number of rows and columns on the screen, but you need to be able to read the data. In order to print *Account Number Inquiry* on the first line and centered we need the line,

screen(1,30) “Account Number Inquiry”,

which will display the above heading in the first row starting at position 30. Any data in those positions on the line when this command is executed will be deleted. Note that if there happened to be something on the screen in positions 1 through 29, it will not be removed. To clear this trash, you can use the line

screen(1,1) “ ” (1,30) “Account Number Inquiry”

or

screen(1,1) “ Account Number Inquiry”

and the result in either case will be only those three words on the first line. This centers the heading in the line.

The first number after the left parenthesis indicates the row where the field will start while the second number represents the column. It should be noted that only the first line will be effected by our screen command. If we use either line above in our program we may run into some difficulties since our screen could already have information all over it on other lines. Our data would still be displayed but so would much of the data that was there before. We will need to clear the screen and this can be done with the line

screen erase

where the combination of these two keywords will remove all previous data from the screen so that we will be ready to display just what we want and nothing more. If we use the line

screen(20,20) “error”

the literal *error* will print on line 20 starting in the 20th column but nothing will be erased before column 20. Adding the line

screen erase(20,1)

before that screen line with the literal *error* will erase whatever is on line 20. Thus

screen erase(10,18)

will erase everything on line 10 starting in column 18.

The complete program would look like the following:

```
program-name: accting
define file acctfile record account-record status acct-status key account-number structure
        account-number integer(9)
        last-name character(18)
        first-name character(15)
        middle-initial character
        street-address character(25)
        city character(15)
        state character(2)
        zip-code integer(5)
        balance signed decimal(6.2)
define error-msg character(60) value “ ”
screen erase
screen(1,30) “Account Number Inquiry”
screen(4,20) “account number:”
```

```
screen(6,20) “last name:”
screen(8,20) “first name:”
screen(10,20) “middle initial:”
screen(12,20) “street address:”
screen(14,20) “city:”
screen(16,20) “state:”
screen(18,20) “zip code:”
screen(20,20) “account balance:”
screen(22,20) “to exit, enter 0 for the account number”
input-number: input(4,36) account-number
        screen(24,1) erase
        if account-number = 0
                go to end-program
        end-if
        read acctfile key account-number
        if acct-status = 0
                screen(4,36) account-number
                screen(6,36) last-name
                screen(8,36) first-name
                screen(10,36) middle-initial
                screen(12,36) street-address
                screen(14,36) city
                screen(16,36) state
                screen(18,36) zip-code
                screen(20,36) balance, mask($$$$,$$9.99-)
        else
                if acct-status = 5
                        screen(24,20) “The account # “ account-number “ is not on the file.”
                else
                        error-msg = “Problem with the account file; program ending – press enter”
                        go to end-program
                end-if
        end-if
        go to input-number
end-program: screen(24,1) erase screen(24,20) error-msg input
        end
```

Some of the statements should be familiar to you. Note first that the title of the account number inquiry program is

```
        acctinq
```

and our second statement

```
        define file acctfile record account-record status acct-status key account-number
                structure
```

introduces the keyword

file,

which we should have had in the earlier programs. It lets us know that we are defining a file. The file status is also included in our definition of the file – we don't need another line for that, though we could have had a separate definition of it – and so are the record layout or structure, and the key of the file, since we are inquiring on the account. As mentioned earlier, all the files in our system are indexed files, so we'll read them with a keyed read, even if we process the file one record at a time.

We clear the screen, but only once, and print the title on line 1 in column 30, followed by all the other headings on the appropriate lines. The next line introduces a new keyword:

input-number: **input**(4,36) account-number

giving the person at the screen the chance to enter a nine-digit account number. The keyword

input

halts all activity until something is entered. If there is input, we clear the error message at the bottom of the screen if one is displayed. It was on the screen a sufficient amount of time. This is necessary for two reasons: first, we need to give the person entering data a chance to look things over; second, it's impossible to read the screen if the data is erased too fast. This reminds me of one of my college professors who wrote with one hand and erased what he wrote with the other – not my idea of a good teacher. When it comes to important messages at the bottom of the screen, a good practice is to leave the error message on the screen until something is input. We have defined

account-number

as a nine-digit integer and whatever is entered to the right of the literal

account number:

has to be a number from 0 to 999999999. If it is, the account number file can be read. You will note that entry for the account number begins at position 36 in row number 4 but if we had omitted the (4,36), it would start in the position right after the colon following the literal. This mention of the specific row and column allows for a space.

You will not be able to enter letters of the alphabet or any special characters into the input field. If you enter 9 for the account number, you would not need to key the leading zeros, since that one digit would suffice. The next line is a decision, which allows the user to end the program, which is done by entering 0 for the account number. Zero is certainly an integer but not a valid account number. As we have pointed out, account numbers have to be greater than 9. Entering 0 terminates the program and forces a branch to the label

end-program

which ends all activity. The next statement

read acctfile **key** account-number

should be familiar. The only difference involves the keyword

key.

This statement takes the value that was entered for the account number, and uses it as a key to read the account file into the structure. The statement could be replaced with

read acctfile

since the definition of the file points to the key as

account-number.

The file status is checked as in our original program. Here we have an indexed read. If the read doesn't find that particular record, the result is a not found condition – status being a 5. If the read is successful, the program will display the values of each field for the record that was read and the user can now enter another account number or exit the program. For invalid account numbers entered, a message will print on line 24 starting in column 20 listing the invalid input. The keyword

else

gives us options so we will either read another record or be at the end of the program. This is another keyword which we have not seen with if statements. We actually cover three cases for the file status, 0, 5 and anything else. The error message

Problem with the account file; program ending – press enter

could result if the file didn't exist or there was some other problem.

You may ask why we need to restrict the account number input to numeric as the read of the file would take care of that. We could take whatever was entered and try to access the file. If the field keyed is not numeric we would get a not found and that wouldn't be a problem, except the operator will have to input another account number, slowing down the input process. If the field is numeric, the record desired may be on the file, but maybe not. Each case would be handled. We could allow

character(9)

but then the user would have to key in leading zeros. Being defined as

integer(9)

is a better choice since it saves keying.

If you have just an if statement or a combination of if and else, you only need one **end-if**.

We could also have written the code for checking the status as

```
if acct-status = 0
        screen(4,36) account-number
        screen(6,36) last-name
        screen(8,36) first-name
        screen(10,36) middle-initial
        screen(12,36) street-address
        screen(14,36) city
        screen(16,36) state
        screen(18,36) zip-code
        screen(20,36) balance, mask($$$$,$$9.99-)
        go to input-number
end-if
if acct-status = 5
        screen(24,20) "The account number " account-number " is not on the file."
        go to input-number
end-if
```

error-msg = “Problem with the account file; program ending – press enter”

and no else statements would have been necessary.

There is one other difference from the first program and that is

balance, **mask**($$$$,$$9.99-).

The minus sign allows for negative numbers to be displayed. For a negative balance, the sign prints to the right of the second decimal digit. If the amount is positive, no sign will print. The negative sign is printed on the right because we have the dollar sign on the left.

A new field is defined by a new keyword in the line

define error-msg **character**(60) **value “ ”**

which says the field is sixty spaces, even though you only see one. This is the same as

define error-msg **character**(60) **value “ ”**.

At the end of the program, that error message is printed. If everything went smoothly, we just print nothing on line 24. Otherwise, an error message is moved to the message and it will be displayed at program’s end. The error message will remain on the screen until the operator presses *enter*.

That is the entire screen program to inquire on accounts and if you think that it is complicated, it does get involved but there are some systems where the whole process is even more confusing. Some methods of getting data to the screen involve escape sequences, which are quite different and mind-boggling until you get used to them. If you’re not familiar with them, you may want to leave for another job. Our language is intended to instruct without being confusing but there will be topics that have some degree of difficulty and unfortunately they just can’t be avoided.

As promised, I offer my thoughts on going paperless. A few years ago while on a contract at a major bank, I saw a report – I forget how often it came out – that used over a box of paper. Strangely, the accompanying summary report was even longer. The requester probably had a few shares of Georgia-Pacific stock. No one said banks were dispensers of sanity. As you can tell, I care for the earth and prefer the paperless route, but not completely. My experience with the backup files of a previous chapter should have convinced you of that. A balance is needed, because of all the possibilities. Our friend Murphy may be lurking somewhere.

Having worked at over a half-dozen software contracts at banks over the years, I saw that those institutions documented everything to death. In the summer of 2011, I needed some documents from a local bank on a few investments. They went back less than six years. The bank found some but not all of what I requested. I was confused and thought that if accounts needed to be purged, they could have been printed out and stored, with an index of the account numbers on some kind of storage disk. The latter would point to the location of each account number. Apparently, I never considered the fire that destroyed the paper stuff, rendering the related disk worthless. Now we know why our parents hid money in out-of-the-way places in their home, so no one could find it, not even their children – ever.

8. Program flow and compiles

So far the two programs that have been discussed have taken an input file and produced output. In each case the input was the account file and the output was a report, even though one was paper and the other was on the screen. We still have to deal with updating a file, but once we do that, we will have covered every possible scenario as to what a computer program can do. We won't get to the update feature for some time but right now let me summarize the way computer programs process line of code.

A computer program will execute one statement after another until finally it hits the end. In our P language, that happens to be the

end

keyword. It can get to different places because of branches to labels, triggered by

go to

statements. These can be conditional, that is based on an

if

statement, but they could also be unconditional. The latter would be seen if we had a group of print statements followed by a

go to

which did not have any

if

statement preceding it. Consider the following statements:

```
If file-switch = 0
        go to next-step
end-if
go to start-read
screen(1,7) "file switch: " file-switch
```

The second

go to

statement is an unconditional branch. Note that the next statement can never be executed because of this

go to

statement. Let's look at another set of statements:

```
If file-switch = 0
        go to next-step
else
        go to start-read
end-if
screen(1,7) "file switch: " file-switch
```

The field

file-switch

is interrogated and if it has a value of 0, program control is transferred to the label

next-step.

Otherwise control goes to the label

start-read.

Because of this logic, the next statement can never be reached as the if-then-else statement has forced a branch to one of two places and it will either get to

next-step

or

start-read.

It can never get to the statement following the

end-if.

This line is *dead code* and there is no reason to leave it in the program. Sometimes people write code this way but the extra line only confuses those looking at the program. On other occasions a program may have been written without any dead code but someone modifies it and in so doing creates these lines of meaningless logic, which can never be executed. Changes were made that forced this scenario and whoever did them should also have deleted the dead code. It's not necessary to keep it there since it will only take someone longer to read the program and realize that this code is unreachable.

Now consider another group of statements:

```
If file-switch = 0
        go to next-step
else
        if file-switch = 1
                go to start-read
        end-if
end-if
screen(1,7) "file switch: " file-switch
```

In this case you will agree that the last line can be executed. This would be true if the field

file-switch

has a value greater than 1. But now suppose that in our program this field will only be 0 or 1. At this point the last line is once again unreachable as before. You couldn't tell this fact unless you had knowledge of what values the field

file-switch

could be.

Let us consider one more set of statements.

```
read acctfile
start:  if acct-status = 0
                print "account number " account-number
                go to start
        else
```

```
                    go to end-program
            end-if
end-program: end
```

At first glance this may look similar to what we had earlier and it appears to be a valid set of statements. Note though that we can never get to the label

end-program

if the file has any records because the variable

acct-status

will always be 0. If the file was empty, then and only then would the program end because the status would not be 0 and the branch to the last statement would be taken. If there is at least one record on the file, the status would be 0 and the print would be done and then a branch to the label

start

would take place. At this point, no new record will be read and the field

file-status

would still have a value of 0 and once again we would print out the same account number preceded by the appropriate label.

What we have is an infinite loop, as the program will continue to print out this same account number with the literal

account number:

preceding it. The program would never end unless we somehow interrupted it. The problem is we need to do the read of the file each time we get to the label

start.

A simple change will get us out of this mess. All we need is to move the label up one line to the read statement and then we would eliminate this looping. Thus our statements become:

```
start:  read acctfile
        if acct-status = 0
                print "account number " account-number
                go to start
        else
                go to end-program
        end-if
end-program: end
```

and now there is no difficulty and the program will eventually end without us having to interrupt it.

This program could also have been written as

```
start:  read acctfile
        if acct-status = 0
                print "account number " account-number
```

```
                    go to start
end-program: end
```

which I prefer since it's fewer lines of code.

We don't want any program to loop indefinitely. You may ask if there is some way to check the lines of the program before we actually run it. There certainly is. There are two possibilities for this. In the P language as in a few others, we write the program and then simply try to run it. If things are in order, all the fields are properly defined, every

if

statement has a matching

end-if,

each left parenthesis is matched by a corresponding right parenthesis and so on, then the program will do what it should. Well it may not do exactly what we want but at least it won't abend and we will have made some progress.

On the other hand, if something is amiss such as an undefined field, then the program will try to run and pause because of that deficiency. At that point there will be some message indicating more or less what went wrong – at least in most languages. If the warning is not specific enough it may list the line with the problem and you can somehow figure out what's wrong, change it and continue to run the program. Of course there could be another different problem and once again you would have the same situation. You could correct it and resume the program and eventually get to the end.

Any program that we write is referred to as source code. Whether the running of the program proceeds as above or in one other manner, it will still be a source program. The other possibility is that we will need to compile the program or source code. This process is nothing more than running the program through a compiler to find out if there are any errors such as we mentioned above. If there are problems, we will get a list of the trouble spots and we can change the code and then recompile the program. Once all the errors are cleared up, the compiler will create what is referred to as object code. This is what is actually executed when we run the program, at least if we compile the source code. If you were to try to view some object code it would probably look like obscenities. However, this is what the computer needs to run your program and object code is sometimes called machine code for that reason.

As far as compiling a program, it sounds complicated but it is just running your source code through another computer program to create object code or find basic program problems. With the P language we don't have to worry about that process but we still need to take care of these errors when we run the program, provided there are oversights in the code. When you get to work with other people in a programming environment and you need to do a compile, you will be given enough information to proceed.

At this point you may think that you are home free if your program compiles and then runs to completion. Maybe so, but there could be logic problems. When you design a program you have a good idea what you want done so you code accordingly. The computer then follows your directions but if you unknowingly have the wrong code, the

computer will still do what you tell it but it may not be exactly what you want. This is referred to as a logic error. What you then have to do if you see that the wrong thing is happening is check over the code and see what is causing the difficulty. That may take longer than it took you to write the program, but you need to do it.

So you may have thought that you would have an easy time but there could be problems. Just remember that the computer will do everything you tell it to do but it is up to you to dictate the proper instructions. This means you need to know the rules of the language and how everything proceeds. The clearer your understanding, the fewer difficulties you will have. When the program doesn't do what you want it to, you have to do some debugging. This is the process of figuring where you went astray.

9. More Modifications

Let's return to our very first program and consider what happens if we have over a hundred accounts on the file. If we run the program the report will have heading lines, followed by detail lines. It will fill up one page and then print the next one without any header lines. What we want to do is change the program so that there will be a title on each page, subheadings as before, page numbers and today's date. Note the program name has been changed. To do this the complete program follows.

```
program-name: acctlist
define  main-heading structure
        print-month character(2)
        field character value "/"
        print-day character(2)
        field character value "/"
        print-year character(48)
        field character(68) value "Account number report"
        field character(5) value "Page"
        page-number integer(3)
define  sub-heading structure
        field character(54) value "account #    last name          first name      mi   street "
        field character(68) value "address          city          state zip       balance"
define  print-line structure
        print-account integer(9)
        field character(4) value spaces
        print-last-name character(22)
        print-first-name character(19)
        print-middle-initial character(5)
        print-street-address character(29)
        print-city character(19)
        print-state character(6)
        print-zip-code integer(5)
        field character(2) value space
        print-balance, mask($$$$,$$9.99-)
define  file acctfile record account-record status acct-status key account-number structure
        account-number integer(9)
        last-name character(18)
        first-name character(15)
        middle-initial character
        street-address character(25)
        city character(15)
        state character(2)
        zip-code integer(5)
        balance signed decimal(6.2)
```

```
define work-date character(8)
define record-counter integer(5) value 0
define page-counter integer(3) value 0
define line-counter integer(2) value 54
define error-msg character(60) value spaces
work-date = date
print-month = work-date(5:2)
print-day = work-date(7:2)
print-year = work-date(3:2)
account-number = 9
read-file: readnext acctfile
        record-counter = record-counter + 1
        if acct-status = 0
                print-account-number = account-number
                print-last-name = last-name
                print-first-name = first-name
                print-middle-initial = middle-initial
                print-street-address = street-address
                print-city = city
                print-state = state
                print-zip-code = zip-code
                print-balance = balance
                line-counter = line-counter + 1
                if line-counter > 54
                        perform print-headings
                end-if
                print print-line
                go to read-file
        else
                if acct-status not = 9
                        error-msg = "There was a problem with the account file"
                end-if
        end-if
        go to end-program
print-headings: page-counter = page-counter + 1
        page-number = page-counter
        line-counter = 5
        print page main-heading
        print skip(2) sub-heading
        print skip
end-program: record-counter = record-counter - 1
        print skip(2) 'the number of records read was ' record-counter
        print skip error-msg
        end
```

You will note that we have a few more structures, one each for the main title, subheading and the print detail line. Most of this should be familiar and you can take for granted that I have all the spacing correct. For the main title line we'll print the date in mm/dd/yy format at the leftmost portion of the line and the page number will be found at the rightmost portion of the same line, preceded by the literal, *Page*. The variable

main-heading

is set up to give us all we need. The first new keyword you see is

field,

which appears to be a variable. It is used mostly for literals or to separate one field from another. The first occurrence of it is

field character value "/"

and this is one position which is in the third column of the line which will always have a value of /. It is the separator between the month and day and the day and year in our date. It occurs twice because we need it twice. We've seen the keyword

value

before. It's used to tell us that this one character field has a specific value. There will be other uses of the

value

keyword, for literals or constants – fields that don't change. Though

field

is a keyword, we cannot refer to it in our program. For example, we couldn't change its value. A few lines down you'll see

field character(4) **value spaces**.

We could have written the statement as

field character(4) **value** " ",

which we used before. Each of the two statements, as well as

field character(4) **value space**

achieves the same result – even though one may not be grammatically incorrect. They represent spaces, a space, nothing, a single blank and blanks.

You will note that

field

occurs a few times in the program and just about each value is different. It represents either a certain amount of spaces for separation or a specific literal and is part of a structure, which enables us to use it. If we need to assign it a value that can change, we have to make it a variable and then we could reference it. Using

field1

or

field2

would do the job, although it would be better to give these fields more meaningful names.

So the beginning of the main title line is a two-character field that is

print-month,

but as you can tell it has no value. We'll give it one in the program and do the same for

print-day

and

print-year,

all based on the current date. If you move down to the line

work-date = **date**

you will see how this will be accomplished. This line has another keyword

date

which represents today's date in the format yyyymmdd. Thus if today is September 10, 2001, the field

date

will have the value

20010910.

Note that not only do we not have to define this variable, we actually couldn't if we tried. What this statement does is assign the variable

work-date,

which we have defined as

character(8)

the value of the field

date.

This is done by the equal sign, which takes whatever is on the right side (today's date in this case) and moves it to the field on the left side. Note that when this move is done, both fields will have a value of

20010910.

It's more like a copy than a move. You have seen the equal sign before but there it was used as a logical operator in conjunction with the

if.

In this line and a few others in the program, this symbol is used as an assignment operator as it gives a field a value. We have another assignment in the line

print-month = work-date(5:2)

which moves the two characters in work-date starting in the fifth position to

print-month.

This is

09,

which happens to be the month. You can then see that

work-date(7:2)

turns out to be the two characters in

work-date

starting in position 7, which are

10,

which is the day. The statement after that takes two characters starting at position 3 or

01,

which is the last two digits of the year. So the four statements of our program after the last define statement get today's date and format the date on the main heading line as

09/10/01.

Tomorrow it will be

09/11/01,

a day that will live in infamy – although I didn't know it when I wrote this book or did the first revision on it.

The statements

account-number = 9
read-file: **readnext** acctfile

we've seen before. A sequential read of the file is being done, starting with the first record and subsequent reads until the end of the file is reached. You may ask if we could begin at the last record and read the entire file backwards with the first record on the file being the last record read. Think about it and I'll provide the answer at the end of chapter 23.

The verb above is for sequential reads, while

read

is used for indexed reads. For reading sequentially, the status is 0 for good reads and 9 for end of file being reached. Anything else is a serious file error. For indexed reads, 0 status is a good read, 5 means the record wasn't on the file anything else means deep trouble.

The next unfamiliar keyword is in the statement

perform print-headings.

It enables the printing of the headings, resulting in going to the label

print-headings,

where each statement is executed until another label is found. The keyword

perform

is different from a

go to

statement in that control will be returned to the line in the program after the

perform.

The **perform** is done and then the file will be read.

The first line of the actual paragraph for printing headings

print-headings: page-counter = page-counter + 1

is another assign statement which takes whatever was in the variable

page-counter,

adds 1 to it and that will be the new value of

page-counter.

If you go back to where that variable was defined, you will note that it initially had a value of 0 so the result after the addition will be 1, which will be moved to the variable

page-number

on our main title line. You expect it to be 1 since this is the first page. This move is done by the line,

page-number = page-counter,

another assignment statement, where

page-number

can be found on the title line.

The next statement is another assignment statement that results in

line-counter

having a value of 5. This is needed because we have to count the lines that we print and when we finish with the main title and the subheading, we will have accounted for five lines. If you think it should be less than that, I'll get around to an explanation shortly. Our next statement

print page main-heading

will go to the top of the page and print the structure

main-heading,

which is the beginning of the report. If we don't use the keyword

page,

the report could start in the middle rather than at the top. This keyword gets us a new page on the printout. The next statement

print skip(2) sub-heading

will skip down two lines before it prints the subheading or structure

sub-heading.

This allows for two blanks line between the main title and the subheading. If you used the print without the

skip

the subheading would print directly underneath the main heading. With

skip(1)

or equivalently,

skip,

the subheading would follow a blank line after the main heading line. From the definition of the heading line,

field character(5) **value** "Page"

results in *Page* followed by a space being printed. That space at the end separates the literal from the actual page number on the title line. This usage is in the main-heading structure as well as in the sub-heading structure. The statement

print-year **character**(48)

will result in the 01 followed by 46 spaces. This centers the title since the assign statement

print-year = work-date(3:2)

will move the

01

to the variable

work-year

and the remaining 46 characters will be filled with spaces.

Getting back to assigning a value of five to

line-counter,

the main heading line, two blank lines, a subheading line and another blank line would have been printed, adding up to five lines. Note that when we print a detail line, we don't include

skip,

meaning there are no blank lines between detail lines, but there is a blank line preceding the very first detail line on every page. Also note that a check is done before we print a detail line to see if it's time for headings. This is determined by checking the variable

line-counter.

First we print the headings and set that field equal to 5. Whenever we print a detail line we increment

line-counter

by 1 in the statement:

line-counter = line-counter + 1

When the counter gets to be greater than 54, it's time for a new set of headings so we'll perform the

print-headings

paragraph and return. Since the counter is initially 54 and we just added 1 to

line-counter,

we perform the heading routine before printing the first detail line, rather than printing the heading after starting the program. Note that once we print the headings we have to reset

line-counter

or else we will have headings for every account number, which we don't want.

We have a slight variation in the line

if acct-status not = 9,

which introduces the logical operator

not.

As we proceed through the file, the status might be greater than 0. A value of 9 will indicate the end of the file. If we get a value other than this, there is a problem, so we will move an error message to the field

error-msg

and end the program.

That does it for our report program. Note that our original program had a few things missing but we took care of those. You can also see that there appears to be two different ways of doing the same thing. If you work on a PC, you will note that this is also the case and most systems do allow a great deal of flexibility. You can choose how you handle a particular situation and either way should get you what you want.

10. Assigning values

In the last chapter we saw the keyword

value

as well as the use of the assign statement, which is accomplished by the equal sign. Before proceeding, let us look at some examples of both these concepts since there are two different ways of accomplishing the same result. The

value

keyword allows us to give a variable some initial value. Thus

define x **integer(**5) **value** 7

results in the field

x

being equal to *7*, while

define y **decimal**(2.2) **value** 3.1

results in the field

y

being 03.10. In the first case

x

consists of five positions with the last being 7, preceded by four leading zeroes. The values *7* and *00007* are equal. The computer will store that number with the leading zeroes. In a similar manner, 3.1 = 03.10, but the computer will actually have 0310 for this field without the decimal point, and it is understood to be there. In our programs we don't need to specify leading zeroes if fields are defined integer or decimal. For character fields we have to supply leading zeroes, if they exist. Consider the following:

define soc-sec-no **character**(9) **value** "45169123"

This will result in that variable being the eight characters above followed by a space which is in the ninth position. As of today, that is not a valid social security number. If we really wanted the first position to be a leading zero as in some social security numbers, our value clause would have to be

value "045169123"

rather than

value "45169123".

Note that this would not be the case if we had defined the variable as an integer since the leading zero would have been assumed.

As we have already seen

define z **character**(5) **value** "Page"

results in

z

being *Page* where the rightmost position is a space. This means that fields that are defined as numeric, whether

integer

or

decimal

will fill the field with leading zeroes where necessary while those defined as

character

will fill the trailing positions with spaces if needed.

This same consideration will apply with assignment statements. Using

soc-sec-no

and

x

as defined above, the statement

x = 3

will result in

x

being *00003* inside the computer, while

soc-sec-no = "111111"

will result in

soc-sec-no

being 111111 , that is, six 1's followed by three spaces. Note that assignment statements involving

character

fields need the quote marks while those dealing with numeric fields will not use them. We could eliminate the

value

keyword and accomplish the same result by using a single assign statement. All we have to do is utilize a group of assign lines just after the last

define

keyword, one for each

value

occurrence. Since we do have the

value

keyword, we have the option of giving a field a value in one of two ways. One is initialization and the other an assignment statement, which often changes the value.

Since this concept is so important in programming, let us consider a few more examples. Assume the following

```
define x character(6) value "Handel"
define y character(4)
```

If the following line is executed,

y = x

x

will still have the value

Handel

but

y

will have the value

Hand,

since the field

y

only has room for 4 characters, it can only accept that many and so truncation will occur. Note that in any assign statement, the variable on the right side of the equal sign will be unchanged. If the first assign is done followed by

x = y

the result will be that

x

will have the value

Hand ,

that is the rightmost two characters will be spaces and

y

will be

Hand,

since that value won't change from the first assignment.

Now let us consider

```
define a decimal(6.2) value 82344.57
define b decimal(4.1)
```

Once the following statement is executed

b = a,

a

will be unchanged but

b

will have the value *2344.5*. Thus truncation will occur on both the left since the variable

b

only allows for 4 digits to the left of the decimal point as well as on the right since there is only room for a single digit to the right of the decimal point. Note that the number that results in

b

is not rounded. You may never run into this situation exactly but may need something similar in order to truncate a value.

We have seen the following before but since we will be using it over and over, it is worth repeating.

```
define line-counter integer(2) value 0
loop:    line-counter = line-counter + 1
         if line-counter > 9
                  go to end-program
         end-if
         go to loop
end-program: end
```

The statement following the

define

will be executed 10 times, that is until

line-counter

is greater than 9. The variable

line-counter

will start out as 0 and

loop: line-counter = line-counter + 1

will result in 1 being added to the field and 1 will now be the new value of

line-counter.

A check on the value of the variable is now done and since

line-counter

is not greater than 9, a branch will be made to the label

loop.

Then

line-counter

will become 2 and the check will be done again and once more a branch is made to the label

loop.

As the program continues to be executed,

line-counter

will become 3, 4, 5, 6, 7, 8, 9 and then 10 and now since

line-counter

is greater than 9, a branch will be taken to the label

end-program

and there is no further action.

This statement at the label

loop

is a way to increment a field, such as we had in our program when we needed to keep track of the lines printed in order to know if a new set of headings was needed. Of course we could increment this field by 2 or 3 or even a negative number if we desired. If our increment statement had been

loop: line-counter = line-counter – 2

note than we would have the following values for

line-counter:

2, 4, 6, 8, and 10. If you think it should be -2, -4, -6, -8, -10 because we are subtracting 2, recall that the variable

line-counter

is defined integer, with no provision for the sign. Thus the first increment will do the subtraction but since the computer has no place for a negative sign, the

line-counter

will be 2 at first. Eventually it will get to 10 and we will branch to the label

end-program.

If we have

```
define line-counter signed integer(2) value 0
loop:    line-counter = line-counter - 2
         if line-counter > 9
                  go to end-program
         end-if
         go to loop
end-program: end
```

note what will happen now. The field

line-counter

will be 0 to start and then -2, which is what we want, and then -4, -6, -8 and -10. But it will not stop there as we will be stuck in a loop since

line-counter

is not getting closer to the value 9, it is proceeding in the opposite direction, becoming a smaller negative number. We could change the lines above by initially setting the value of

line-counter

to 10 and changing the decision line to

if line-counter = 0

and now the endless looping will not take place.

Consider the following

```
define j character(11) value "candlelight"
define k character(6)
define m character(5)
k = j(1:6)
m = j(7:5)
```

If we look at the variables after the assign statements,

k

will be *candle* and

m

will be *light*.

That's because we're getting pieces of a string of characters from one variable. The expression

j(1:6)

will result in the characters in the variable

j

starting in position 1 for 6 positions being moved to the variable

k,

so it becomes the value *candle*. For the variable

m

5 characters starting in position 7 will be moved to

m,

which happens to be the value *light*. This may come in handy some time and it also means that we could eliminate the

structure

keyword since we could extract any field we want from

account-record

by using an expression like

last-name = account-record(10:18)

which will give us the last name from the record. However, we won't eliminate the keyword

structure

as it just might save us some time and work. As in many situations, we have some freedom and can use whichever method is more beneficial.

11. Updating fields

Before we proceed into updating fields in the account file, let us talk about some possibilities. We could create a second program that could be used for updating the fields or we could incorporate the options into the inquiry program. The advantage of two programs is that we would keep each process separate and thus the logic for each program would be less complex. On the other hand, many of the same processes will be in each program so perhaps a single program would be better. With the latter option, all the code would be in one place, which means that if changes have to be made they would be done in a single place. If we had two programs then we could miss changing one when there were changes to the other.

Either having one or two programs is an acceptable choice. Obviously any system should be designed to make the person using it as comfortable as possible and it should be able to be easily maintained. The choice is not an easy one and having both *user-friendly* and easily maintained systems may not always be possible. One feature may have to sacrificed for the other and much thought needs to go into how things will be handled.

In our system, account numbers will be generated by the computer. They could have been created by people at the computer as they entered all the fields. The reason for using the machine and not people is because this approach will be easier for the user insofar as the computer will have knowledge of what numbers are available, as they are needed. If ten people are entering data at ten different places for the account file and need to put in a new account number, they may try one only to find that that value has already been used. This effort could result in the person inputting information getting delayed because he cannot find an unused account number.

What this choice means is that to accommodate this decision, we will have to spend more time developing our program. There will be more work for the programmers, but this will be well appreciated by the users. Since we do have computers, there's no sense in not taking advantage of their power. In the process we should make every effort to guarantee that our systems are maintained as easily as possible. We will talk more about the assignment of account numbers at a later time.

Using a single program, our old inquiry program will now give us the ability to change the values of certain fields on the account file. For now, there will be no update of the account number field because this would equate to a delete of a record and an add of a new one. Only the fields last name, first name and balance will be allowed to be changed for discussion purposes. The other remaining fields could be changed in the same manner so there is no need to show all of them in this program. You have to do it yourself but you will get the idea from these three fields alone.

The beginning will not change as the user will enter the account number and the account file will be checked to see if it exists. Entering a 0 will enable the user to exit. This is the same as in our original inquiry program. Assuming the record exists, the fields on the record will be displayed along with an additional message at the bottom of the screen and there will be numbers to the left of the three fields that can be changed. In the event that no changes are to be made, the operator can enter a 0 and now another account number can be entered. All three fields can be modified or as few as one, but once a

change is made the account file will have to be reread in update mode to see if anyone else has that record. If so, the record cannot be updated at this time. If the record is available, the program will then write the account record with all the changes. Whether this is done or the record is busy, the program will then return so that another account number can be keyed.

This doesn't sound too difficult but it is more complex than described here. To begin with, we don't want someone to enter characters into either name that are not allowed there, such as $ or %. Thus we should try to eliminate this possibility and warn the operator of this error with some kind of message. Fortunately by defining the account balance as

signed decimal

we won't have to worry about what is keyed since any character that is not acceptable as this type of field will not be allowed by the input statement. We can check for certain characters in either name, but the user could enter "h-o-s-e-r-s" and that would be fine, with no objection. We will assume that the operator knows what he is doing. In many companies, that belief can be dangerous.

Most of the inquiry / update program will look familiar but there will be a few new keywords and some of the logic may be troubling or puzzling or both. After going through it, you should have a good grasp of what an update is all about. Assuming the operator enters a valid account number of

391842091,

the new screen will look like the following:

Account number inquiry / update

Account number: 391842091

1. last name: Smith

2. first name: John

middle initial: L

street address: 3910 Main Street

city: East Aurora

state: NY

zip code: 14052

3. account balance: $27.89

key 1 (last name), 2 (first name), 3 (balance) or 0 (next)

To change the first name, enter 1. To update last name, enter 2 and enter 3 to update the account balance. Entering 0 will give you the opportunity to enter another account number. The new program will now turn into the following:

```
program-name: acctupd
define  file acctfile record account-record status acct-status key account-number structure
        account-number integer(9)
        last-name character(18)
        first-name character(15)
        middle-initial character
        street-address character(25)
        city character(15)
        state character(2)
        zip-code integer(5)
        balance signed decimal(6.2)
define name-string structure
        field character(26) value "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        field character(30) value "abcdefghijklmnopqrstuvwxyz '-."
define field-no integer
define valid-input integer
define update-switch integer
define new-last-name character(18)
define new-first-name character(15)
define new-balance signed decimal(6.2)
define error-msg character(60) value spaces
screen erase
screen(1,25) "Account number inquiry / update"
screen(3,20) "account number:"
screen(5,17) "1. last name:"
screen(7,17) "2. first name:"
screen(9,20) "middle initial:"
screen(11,20) "street address:"
screen(13,20) "city:"
screen(15,20) "state:"
screen(17,20) "zip code:"
screen(19,17) "3. balance:"
input-number: screen(21,1) erase
        screen(21,1) "enter the account number or 0 to exit"
        screen(3,36) input account-number
        screen(24,1) erase
        if account-number = 0
                go to end-program
        end-if
        read acctfile
```

```
        if acct-status = 0
                screen(3,36) account-number
                screen5,36) last-name
                screen(7,36) first-name
                screen(9,36) middle-initial
                screen(11,36) street-address
                screen(13,36) city
                screen(15,36) state
                screen(17,36) zip-code
                screen(19,36) balance, mask($$$$,$$9.99-)
                new-last-name = last-name
                new-first-name = first-name
                new-balance = balance
                update-switch = 0
                field-no = 9
                perform get-change until field-no = 0
                go to input-number
        else
                if acct-status = 5
                        screen(24,20) "account number " account-number " is not on the file."
                        go to input-number
                else
                        error-msg = "account file read problem; program ending – press enter"
                        go to end-program
                end-if
        end-if
get-change: screen(21,1) erase "key 1 (last name), 2 (first name), 3 (balance) or 0 (next)"
        screen(21,60) input field-no
        screen(24,1) erase
        valid-input = 0
        if field-no = 1
                perform input-last-name until valid-input = 1
        else
                if field-no = 2
                        perform input-first-name until valid-input = 1
                else
                        if field-no = 3
                                screen(19,36) input new-balance
                                update-switch =1
                        else
                                if field-no = 0
                                        perform update-check
                                else
                                        screen(24,20) field-no " is invalid"
```

```
                                end-if
                        end-if
                end-if
        end-if
update-check: if update-switch = 1
                        read acctfile update
                        if acct-status = 3
                                screen(24,20) “that record is unavailable – no update done”
                        else
                                if acct-status > 0
                                        error-msg = “update problem; program ending – press
                                                enter”
                                        go to end-program
                                else
                                        balance = new-balance
                                        first-name = new-first-name
                                        last-name = new-last-name
                                        write acctfile
                                        if acct-status > 0
                                                error-msg = “update problem; program ending –
                                                        press enter”
                                                go to end-program
                                        end-if
                                end-if
                        end-if
                end-if
input-last-name: screen(5,36) input new-last-name
        screen(24,1) erase
        if index(new-last-name, name-string) = 1
                valid-input = 1
                update-switch = 1
        else
                screen(24,20) “invalid characters in the last name – try again”
        end-if
input-first-name: screen(7,36) input new-first-name
        screen(24,1) erase
        if index(new-first-name, name-string) = 1
                valid-input = 1
                update-switch = 1
        else
                screen(24,20) “invalid characters in the first name – try again”
        end-if
end-program: screen(24,20) error-msg input
        end
```

To start with, we have the new variables,

field-no,
new-last-name,
new-first-name,
new-balance,
update-switch,
valid-input,

and the structure

name-string.

To indicate which of the three fields will be changed, we need

field-no,

a one-byte integer. The three fields that are defined beginning with the characters *new* are the fields that will be used for the changes we make. The variable

update-switch

is needed to decide if that is to happen. If the field

valid-input

has a value of 1, this indicates that the input is fine for last name or first name. The structure

name-string

consists of all the characters that are acceptable for either name and we will see how it is used in conjunction with the keyword

index.

The initial process is the same as we found in the inquiry program insofar as an account number can be keyed or the number 0 can be input to exit. The resulting screen is a bit different in order to allow for the updating of the three fields. At this point the user can now get the information on another account by entering *0* or she can enter *1* to change the last name, *2* to change the first name or *3* to change the balance.

Let us proceed to the assignment statements after the line

screen(19,36) balance, **mask**($$$$,$$9.99-).

The first three have to do with the values for the changed fields and we will get back to them later. The next line sets the variable

update-switch

to 0 because at this point no update will take place. When it should occur, we'll set it field to 1, which happens whenever one of the three variables is correctly modified. The next field

field-no

is set to 9 but we could have used any value greater than 3. Note that we can't use 0 because if we did, then the next statement, the perform, wouldn't be executed. This would just be an inquiry program with a great deal of dead code. This statement

perform get-change **until** field-no = 0

will be done until the variable

field-no

has a value of 0. It will have that value if the user either doesn't want to update any of the three fields or has done one or more of them and is now finished with the changes, which means he is ready for another account. The paragraph

get-change

is a procedure where the label begins it and it ends at the statement just before the next procedure. The line displayed on the screen starting in position (21,1) is informational as it tells the user what can be input. You can see that there are two different messages displayed on line 21. The next keyword

input

forces the operator to enter 0, 1, 2 or 3 depending on which field is to be modified, and this will be just to the right of the message on that line. The variable

valid-input

is assigned the value 0 which will be used by the two procedures for the first and last names. We don't need it for anything else but it will be set to 1 if the names keyed are satisfactory. The next few lines

```
if field-no = 1
        perform input-last-name until valid-input = 1
else
        if field-no = 2
                perform input-first-name until valid-input = 1
        else
                if field-no = 3
                        screen(19,36) input new-balance
                        update-switch = 1
                else
                        if field-no = 0
                                perform update-check
                        else
                                screen(24,20) field-no " is invalid"
                        end-if
                end-if
        end-if
end-if
```

will enable the fields to be changed. We'll stay in the procedure

get-change

until the operator keys a 0. A value of *1* for

field-no

means that the procedure

enter-last-name

will be executed until the name entered is acceptable. We will get to that procedure in a minute but let us consider the rest of the procedure that we are in. A value of *2* transports

us to the procedure for entering a change to the first name and we won't leave it unless we have entered valid characters.

If

field-no

is *3*, the new balance can be entered. It can positive, negative or 0 and because of the definition of

new-balance,

letters of the alphabet and special characters are not allowed. The combination of the keyword

input

with the definition of the field mean that only a limited number of characters are acceptable here, namely integers, the negative sign and the decimal point. Note that the operator needs to enter the decimal point if the amount involves cents, otherwise the number will be assumed to have no decimal digits. We could have gotten around keying the decimal point by allowing movement to the decimal portion of the amount with a tab. However, since either way involves a keystroke, we chose the decimal point. The balance entering process won't display any error message if the operator keys in a letter of the alphabet, but the cursor will return to the 19th row and 36th column for another try at it. If the user feels stuck on the field, all she needs to do is enter *0* for it, even though that's probably a wrong value.

The next statement asks if

field-no

is 0. Since this procedure is executed over and over, if *0* was keyed it could mean either that at least one field was changed or perhaps none at all so we need to determine if we have to update the file. That is why we perform the procedure

update-check.

There is one other possibility in this procedure. Someone could have entered a *4* or *7* here for

field-no,

either of which is unacceptable. If that happens, we will let them, but notify them by the message that actually prints the value they keyed. So if they keyed *4* they will see at the bottom of the screen

4 is invalid

and now the program goes back to the beginning of the procedure for more input. Perhaps now they will key either 0, 1, 2 or 3 and nothing else.

In any event, we have covered all the lines in the paragraph

get-change,

so let us proceed to the paragraph

enter-last-name.

The first statement

input-last-name: **screen**(5,36) **input** new-last-name

will allow the user to key the last name just to the right of the field label on line 5. The next line

screen(24,1) **erase**

will clear the entire line 24. If you just wanted to clear the part of the line beginning in column 41, you would use

screen(24,41) **erase**

which does that. The reason we need this line is to clear a previous error message in the bottom line. When we print an error message, as stated earlier, we have to make sure that we don't delete it too soon. After all, the user should have a chance to see and read it. Otherwise, why even print it?

The way I determine when to clear an error message is to always do it after the next input statement. Since this stops everything until something is keyed, this assures that it will stay on the screen long enough. So what you need to do is print the error message and have it cleared after the next input statement encountered in the flow. If you have a statement to erase the message line right after each

input

statement, you shouldn't have any problems.

The next line

update-switch = 1

sets

update-switch

to 1, meaning that we need to eventually update the account file because a change has been made to last name. We will also set this variable to 1 if we change balance or first name. The next line

if index(new-last-name, name-string) = 1

uses the new keyword

index

which checks to see if every character found in the variable

new-last-name

can be found in the variable

name-string.

If you recall,

name-string

is nothing more than the entire possible set of values for the last name, letters of the alphabet and certain other special characters. Note that the space is included, along with the hyphen, the period and the single quote.

So if the characters keyed into the variable for the changed last name can be found in this string of values, the result will be 1. If there is even one character input for the name that is not in

name-string,

the value of this expression will be 0. This is how we restrict entry to specific characters.

If the data entered satisfies our requirements, this means that we have valid input, so we set the appropriate variable to 1 by the statement

valid-input = 1

and we are through with the paragraph

input-last-name.

Otherwise we give the user a message with the line

screen(24,20) "invalid characters in the last name – try again"

and the operator has to try again to input a correct name. Note that our code will not restrict the person from entering ten periods for the name, which is not correct but will pass the test. The paragraph

input-first-name

is very similar to that of the last name. Had we allowed changes to the street address, using

name-string

we'd have fallen short in out checking process because we need a few more characters such as numbers. These are needed for making up a valid street address, and there may be a few more acceptable characters that we have to add. Using

name-string

as is will result in *123 Main St.* being rejected.

The

write

statement should always work but we put in the check just in case the value of

acct-status

returned is not 0. The reason why we don't anticipate a status other than 0 is because the account number that we are about to write is what we read earlier. However, there could be other problems with the account file. Let us suppose the data base analyst accidentally deletes the file while we're in the middle of changing the record. At that point the write of the new record would fail. We would take care of that very situation by the error message along with the ending of the program since we cannot proceed further. We trust the file won't get deleted by mistake, but accidents do happen. Don't forget, we are dealing with human intervention.

Getting back briefly to entering the names and balance, note that the person at the keyboard might get stuck entering any of the three fields. To solve the problem, he could enter

John

for the first name,

Smith

for the last name and

0

for the balance. These would probably be incorrect but at least the person would not be hung up. We don't want the operator being stuck in a loop and never escaping. A better idea would be for the user to note the original three values and key them when stuck. Rewriting the record with the old values won't cause any change – only updating the counter for records updated.

We next look at the paragraph

update-check,

which we will arrive at when the operator keys in 0 for field-no. This means either the screen displayed the data for a specific account and the user decided she wanted to input another account number without any changes to this one or she made at least one modification and now is finished. In the latter case we have to update the account record.

The first statement

update-check: **if** update-switch = 1

determines whether we need an update or not. Recall that we set this field

update-switch

to 1 when a change was made and if there were no modifications, the value of this variable would be 0. If

update-switch

is 1 we will have to read the account file for this specific record and lock the record so no one else can have it for updating. The new keyword

update

will do just that in the line

read acctfile **update**

and we have seen this type of statement before except for the

update.

We now check the file status in the line

if acct-status = 3,

which will be the case if someone else has the record for updating. We won't be able to do the update at this time. This means that the operator will have to try again by making the changes later. If someone else is just reading the record, the status will not be 3 and we'll be able to proceed. A status of 3 then produces a message by the statement

screen(24,20) "that record is unavailable – no update done"

and this will then get us back to enter another account number. I'll spend a bit more time later on a record being busy and unavailable for update. Both reads in the program are indexed reads since we're after a specific account number. In any read of this type, the status will never be 9 – indicating the end of file condition – even if the record read is the last record in the file.

The next lines

```
if acct-status > 0
        error-msg = "update problem; program ending – press enter"
```

mean that we tried to do a read but somehow it failed. Since we were successful before as we did display the data earlier, you might think that it should be successful now and it really should. If the file status is 0, it means we got the record for updating and the next lines

```
balance = new-balance
first-name = new-first-name
last-name = new-last-name
```

move the new values to the record before we write it. You might think that if we didn't change first name the old value would be wiped out since

new-first-name

had nothing in it. We prevent this situation by the three statements in the program just after the beginning

```
new-last-name = last-name
new-first-name = first-name
new-balance = balance,
```

which move the old values of the names and balance to their corresponding new fields. Thus a change to any of the fields will result in the new value but no change will find the old value there. It won't hurt to move the old value if it didn't change before we write the record. We could handle this another way by keeping track of the fields that change and only move them to the appropriate fields when they change but that would mean more variables and

if

statements and our approach will eliminate that.

The next lines

```
write acctfile
if acct-status > 0
        error-msg = "update problem; program ending – press enter"
        go to end-program
end-if
```

accomplish the rewriting of the record. This is done by writing the record from a specific layout or structure, as given in the definition of the file. This keyword works similarly to the read keyword. The modified record is written – rewritten, for you purists – but note that at most three fields were changed. Once more we check the status. If there is a problem we will not only not do the update, but we will also terminate the program because there is a serious problem, moving an error message to be displayed on the screen. If the update is successful, the operator can enter another account number.

All the statements have been covered so let me talk about the approach to updating as well as records being unavailable. There are many ways to accomplish the same results in our program. We could have initially opened the record for update but we chose to do it only if an update was necessary. Doing it initially means we are locking others out from the record when we may not even be updating it so our method will have it frozen for a shorter period of time. Some systems lock the entire file, which we are not doing. Our approach means records will be more available. We could also have locked the record just before we made a change to one of the fields but the fact that the operator is keying data and may have to change another of the three fields implies that the record will be unavailable a great deal longer than if we were to use the approach we take. Our method will only have the record unavailable to others for update for less than a second.

It is true that one of the other ways of reading the record for update may have made the programming easier but we really want to avoid the situation where the record is busy and can't be updated. Our means of getting the record for update should help to

virtually eliminate the possibility of busy records and if other programs did the same thing unavailable records should be very rare.

Of course a record may be busy and we could add code to our program to change the situation somewhat. We could put some kind of pause in the program of ten seconds or so and then try again. This may then make the record available. This will be discussed in chapter 16. Another approach would be to somehow save the account number of the busy record as well as the changed fields and then when the program is about to end go through a paragraph to attempt all the unsuccessful updates once more. We may succeed in some or all of the records but the ones that still couldn't be updated would somehow have to be made known to the operator.

12. Programming standards

Before proceeding, it's time for a test. It's actually an exercise to challenge your brain. The problem is this. We have a balance scale and 8 pills that are all the same color. One of the pills is poison and the others are plain aspirin. The poison pill weighs slightly more than the others and can only be discerned by using the scale. How do you positively determine which is the bad pill in exactly two weighings? A weighing is defined as loading a set number of pills on each side of the scale and making a determination. Think about it and I will reveal the answer at the end of the next chapter.

As I have pointed out, we have certain rules in our P language. These have to do with keywords and their meaning, the way we define variables and use them and how we put together keywords and variables to form meaningful statements to the computer. From the sample programs you will note that I indented certain lines but that was not really necessary. For example the statements

```
if acct-status = 0
        go to read-account
else
        go to end-program
end-if
```

could just as well have been written as

```
if acct-status = 0
go to read-account
else
go to end-program
end-if
```

and the program would have functioned properly just as with the first set of lines. The indentation is done for readability, so that someone looking at the program can more easily follow the logic of the program. I also mentioned that instead of calling the variable

last-name

for the name of the person corresponding to the account we could just as well have used the variable

x.

It's just that the latter would give us no clue as to what it represents without further digging into the program. The variable

last-name

is more meaningful.

Earlier I mentioned that one of the first languages I studied in graduate school was APL. In that language you could eventually reduce any program to a single continuous statement. It might extend over one line but one statement would do. If you wrote the program as a series of statements and thoroughly tested it so that you were completely

satisfied that it worked and then converted those statements to one single entity, that might be fine. But if you had to come back six months later to change it, or if someone else wrote the program and you had to modify that single statement condensation, I'm sure you wouldn't be too pleased.

Our goal in programming is to make things understandable and easily modified. I gave some thought to a certain approach to the update program of the last chapter and decided that my original design would be too confusing and involve more variables than I really needed. As a result I came up with a better way of doing things that would be more easily comprehended. It's true that the other approach would have worked but why not have a design that works and can be both understood easily and modified with few repercussions. Even if you do your best in simplifying a situation, it will still have enough complexity so there is no need to make it more mystifying.

Every programming language has certain rules but there are other guidelines that can be taken to make life easier when we have to make changes to a program. The idea of indentation is one and structured code is another. Some languages or companies have a rule that

go to

statements aren't allowed. If you ask how you can program without it, there are certain situations that can't be avoided because of specific keywords and processes. In general you can somehow replace the

go to

with a

perform

statement. I argue against this if the latter is more difficult to understand than the use of the

go to.

First and foremost the goal of any program is simplicity.

Another guideline has to do with repeating lines of code. If you have five statements in a program that occur in two or three different places, why not put them into one procedure and simply perform that procedure when it is needed. This approach will accomplish two things: first, your program will have fewer lines of code and second, if you have to modify those lines, the change will be in a single place rather than two or three. Changing the code this way means that you won't change the code in one area while forgetting to do it in the other area since it is only in one place. As far as the number of lines of code goes, generally speaking the fewer lines of code you have in a program, the more easily can it be maintained.

I will talk more about guidelines as we progress but for now let us look at more examples of the index keyword. Consider the statements

```
define x character(8) value "01234567"
define y character
define z character(3)
define answer integer
```

and the statements

```
y = "8"
answer = index(y, x)
```

which will result in

```
answer
```

with the value of 0 because the character *8* is not in the string

```
01234567.
```

Before proceeding recall that

```
integer
```

represents a one position numeric field just as the use of

```
character
```

for middle initial in an earlier program involved a single character.

The statements

```
z = "222"
answer = index(z, x)
```

will yield a value of 1 for

```
answer
```

since each character in the variable

```
z,
```

namely the 2 three times, is in the variable

```
x.
```

If we have the statements

```
z = "70"
answer = index(z, x)
```

note that the result will be 0. Certainly the *7* and the *0* are in the variable

```
x
```

but the third character is not. The last character is a space since

```
z
```

is defined as 3 characters and the statement

```
z = "70"
```

results in

```
z
```

consisting of three characters with the last one being a space. Since the space is not found in the string

```
x,
```

the result of the

```
index
```

statement is 0. If you say that z is only two positions long since it has the value "70", I need to remind you that previously we had defined z as a field having 3 characters.

What will the statements

```
z = "70"
answer = index(x, z)
```

yield? What we are asking is if the string

01234567

can be found in the string

70.

Well the 0 and the 7 can but that's about it so the result for

answer

is 0.

What will the statements

```
define x character(8) value "01234567"
define y character value "2"
define z character(3) value "789"
define answer integer
answer = index(index(z, y), x)
```

yield? First note that

index(z, y)

will be evaluated first and the result will be 0 since none of the characters *7*, *8* or *9* can be found in the string

y,

which has a value of 2. Thus the line reduces to

answer = **index**("0", x)

and this is asking if 0 is in the string

x.

It certainly is so the result is that

answer

winds up with the value 1. This is a farfetched example that you probably will never encounter but if you understand it, you have a good grasp of the

index

keyword.

For your assignment, assume your boss likes to get involved in what you're doing – like constantly looking over your shoulder. This may seem like an outlandish request, but just pretend you're working in the department of information technology – or whatever it's called today – at a corporation in America. The manager asks you for this:

Write a program that will allow someone to set up some accounts for the account number file. The user will enter last name, first name, middle initial and an account

number will be generated from the system. A report listing all these fields is required. Later, someone else can key in the other data for these file records.

As you can see, the specifications aren't very detailed, but at least they're not written on toilet paper. If so, I'd refuse the request if my manager believed in recycling. The solution will be presented in a later chapter and illustrates what has to be done to add records – in some small way, you can figure out the rest – to a file.

13. The zip code file

One thing that bothers me is unnecessary work. If I'm at a web site and ordering some stuff to build a time machine, I'll probably have to enter some address information so they'll know where to send the materials. This includes street address, city, state and zip code. I can understand entering the street address, but wouldn't just keying the zip code relieve me of inputting the city and state? This is so because each zip code is tied to one specific state and city. With that in mind, this means that our account number file need not have all three of these fields on file records.

Sly in accounting wants a simple online program – not the one we talked about earlier – to list account number, last name, and the address fields. The following program will do that using a new file, the zip code file. Our account number file no longer carries the state and city, so we have to get it from the zip file. The advantage of this extra read could be used in adding or changing records to our main file – especially if we changed city, state or zip code. Any specific change of this type could cause problems since modifying the city or state could mess up the zip code. Changing the city from Buffalo to Boston means that the zip that you had on the file is not appropriate for Boston. This means that a change to the city needs a modification to the zip code and perhaps to the state as well. Obtaining city and state from the zip code file can make things more precise and also save work for the operator – and the programmer, as well. Another advantage of accessing the zip code file is that the operator can't input an invalid zip code on an add or an update.

You might say that the extra reads of the zip code file may slow down the system but file access these days is quite fast so it shouldn't be a factor. If it is though, the city and state could be stored on the account file since disk space is so cheap and available and this may be preferable to extra I /O. It really depends on your system. You will note that the accuracy of the data on the zip code file is very important and someone is responsible for it. Once again the data is in the hands of individuals and we will assume some integrity on their part.

Another possibility is to put the zip code data into a large table, which our program can access. This method of getting information will be much faster than getting it from reading a file since the approach is internal as we have the city and state available already and need not do any I/O to an external file. Once again the table will change but many systems keep up with this change because the table is dynamic. This means that we bring it into our program as it currently exists so that all the latest changes to it are included. I'll talk about a topic related to this in another chapter when I get into *copy members*.

This program is named after the requester.

```
program-name: acctsly
define file acctfile record account-record status acct-status structure
        account-number integer(9)
        last-name character(18)
        field character(58)
```

```
        zip-code integer(5)
        field character(10)
define  file zipfile record zip-record status zip-status key zip structure
        zip character(9)
        city character(15)
        state character(2)
define error-msg character(60) value spaces
screen erase
screen(1,26) “Sly’s account number inquiry”
screen(4,20) “account number:”
screen(6,20) “last name:”
screen(8,20) “city:”
screen(10,20) “state:”
screen(12,20) “zip code:”
screen(22,20) “to exit, enter 0 for the account number”
input-number: input(4,36) account-number
        screen(24,1) erase
        if account-number = 0
                go to end-program
        end-if
        read acctfile
        if acct-status = 0
                perform get-city-state
                screen(4,36) account-number
                screen(6,36) last-name
                screen(8,36) city
                screen(10,36) state
                screen(12,36) zip-code
        else
                if acct-status = 5
                        screen(24,20) “the account # “account-number “ is not on the file”
                else
                        error-msg = “account file read problem; program ending – press enter”
                go to end-program
        end-if
end-if
go to input-number
get-city-state: zip = zip-code
        city = spaces
        state = spaces
        read zipfile
                if zip-status = 5
                        screen(24,20) “that zip code ” zip “  is not on the file”
                else
```

```
                    if zip-status > 0
                        error-msg = "zip file problem; program ending – press enter"
                            go to end-program
                        end-if
                    end-if
end-program: screen(24,1) erase screen(24,20) error-msg input
        end
```

Everything should look familiar since we have no new keywords. The account number file has been defined a bit differently, but it's still the same file. We're using a new file that has three fields, the city, state and zip code. That's the zip code file. Any read resulting in a not found record for either the account number or zip code file, will produce an error message, but allow another try for input. A problem other than that ends the program with an error message. Variables won't be displayed on the screen as requested by Sly if there is a zip code problem other than a not found. That's because the branch will be done to

end-program

first.

It's time to return to our brain challenge of the previous chapter. To begin with, if you divide the tablets into 4 on each side of the scale, you will need three weighings, and not two. So why not place three pills on each side of the scale. If they do balance, it means that the poison pill is one of the two remaining so just put one each on the balance and now you know which is the bad one. However, if the three versus three match results in one side weighing more than the other, you know that the poison pill is on the side with the heavier ones. From those three take any two and place them on the scale. If one weighs more than the other, you have the culprit but if they weigh the same, the poison thing is the pill you left out of the last three. Thus you have found the undesirable pill in a mere two weighings.

14. Programming creativity

That last challenge with the pills was meant for two purposes: first to keep the creative juices flowing and second to point out the fact that sometimes the normal way of doing things may not always work. We may have to find a different way of accomplishing a task. Even as we progress from day to day you will note that methods of getting a task done on the computer will change from what they were. It's true that the technique of yesterday may still work today, but it may be beneficial to improve on the method just as the technology itself has been improved.

The obvious reference here is to the days when we had more limits on what we could do as computer programmers. We may have had space limitations, which forced us to think in a different way, or there may have been a limit on the number of variables in a program or the size of a table, which meant that the approach we thought about couldn't be used. We would have to come up with an alternative method. A particular situation could be handled in two or three ways but perhaps only one of these would allow us to get certain tasks finished in a required timeframe. Many of these early restrictions are gone now, but some are still with us.

I recall a specific instance when I was working on a modification to a program to add checking for a new transaction. Unfortunately the software wouldn't let me do this by increasing the table to accommodate the additional three characters. It seems there was a limit on the size of this field and it had been reached. Fortunately there was no repercussion if I added more lines of code, but there could have been. The way the program worked before with this table was that it started at some position in the string of values and obtained that number and the next two. If the string was

001002003004,

a pointer would start at position 1 for a length of 3, with the result being

001.

This value would represent a transaction number. We could then bump up the pointer by 3 to get 4 and then take the string starting at position 4 for a length of 3 and the result would be

002.

These three position fields would represent transaction numbers and by looping through the values in the same fashion, we could check to see if the three digit number we had was a valid transaction.

My task was to add another three-digit number that represented a new transaction to this string. Unfortunately the limit on the size of this variable had been reached. Since my new transaction was

010,

what I did was to add a slight modification to the program that checked the value of the pointer and if it was 1, I would only increase it by 1 rather than by 3. Then from that point in the string if I took the next three digits I would get

010,

which would account for the new transaction possibility. I would then check the pointer and if it was 2, I would increase it by 2 and then proceed. In this way I would still have available all the possible values I had originally as well as the

010.

Thus I added logic but didn't have to increase the size of the variable, which I couldn't do anyway.

I talked about adding a read to the zip code file but also mentioned that the added read of the new file could result in a slightly longer run time. This could be remedied by storing the city and state on the account file even though it could also be found on the zip code file. If this still caused a problem because of the lack of space, I suggested putting the zip code data into a more easily available table, which would mean less time needed for file access since you wouldn't have to read the zip code file, and less space required on the account file since you wouldn't have to store city and state on it. In making any of these decisions, you need a good knowledge of the capabilities of your system as well as what your options are.

You will run into challenges of all sorts despite the improvements of computers in what they can do and how fast they can do it. No matter what you do, you may not be able to speed up some process, especially if it depends on reading a file. You will have to make some decisions such as getting rid of old records by moving them to a history file and that could save you some time. This would mean less file access and it might save the day for you. Another consideration may be to change the way the file is accessed. This may not work so you will have to come upon another solution. Despite the speed of today's computers, it still will take some time if you have to read through 10,000,000 records.

Some time ago I added logic to a program to do some type of string manipulation using some keyword. I forgot what it was or involved but I do remember that it took a great deal more time than I wanted it to despite the fact that it worked. I spent some time thinking about it and tried another method to get the same results. It also worked and fortunately only took a fraction of the time of the initial approach. You will run into situations just like these and your brain will be challenged.

As you know, there are many different ways in the world of PCs to accomplish a given task. That is true in programming, as I have already pointed out. The question might be what is the best way to accomplish a specific task? Obviously if you can do it by a simple process and keep the user happy as well as provide a program that is easily maintained, you have found the best way to get the job done. As you might guess, all three of these may not always be done together. So if you have a chance to provide someone who requested a program more than she wanted and at the same time the program can be easily maintained as well as not that much of a headache in producing, all the better. Intelligence and consideration are necessary.

Recall not too long ago I mentioned that the update program would not allow for changing an account number as that would be done by a deletion and an add. Since people close accounts out every so often, we will still need to be able to delete an account. Once we have added that functionality along with that of adding an account, which we shall get to shortly, we will have all we need to maintain the account file.

Different circumstances in other systems may require the ability to change an account number, and you will encounter these challenges in your assignments.

As far as the account number goes, you may wonder how the system generates it. One way could be to generate a random nine-digit number and then read the account file. If the number created was not on the file we would then have a new account number to use. Initially this process may be all right but once the account file had a few records, it just might take a few random generations and corresponding reads of the file to find a free number. The process of generating a nine-digit number takes no time at all but reading the file a few times would.

Though this may work, we will use a different approach and eliminate some of the file accesses. We will still have a file to read but at most we will only read one record because our file will only have one record in it. That record will turn out to have the next available account number. What will happen is we will read this file and take the value on it for our account number and then 1 will be added to that number. We will then write the record. If someone else needs an account number, they will read this same file and get a number and the process will be repeated. Since each read of this file will be in update mode, the record will be locked so no one else can get it until we are through. The record they read will then have another account number and duplication should be eliminated.

What kind of problems could arise using this method? Suppose someone got into the program to add a record, got a new account number and then decided they didn't want to have that record at all. Once in the program we may not give them the chance to delete the record but there would be another program for deleting closed or unwanted accounts. Even if we allowed the deletion in the add program, you can see that eventually we would have gaps in the file. That is to say, there would be numbers in the sequence that would not correspond to a valid account number. This may not be a problem at all as we have almost a billion numbers to choose from. However, if there were many deletions and the account number was only 6 digits and we had more customers that we anticipated, the possibility exists that we could run out of valid account numbers.

To get around this potential problem you could solve it by writing another record to the next available account number file every time a delete was done. This number could then be reused. The process would involve more complications in the logic as far as accessing the file for this number and it would cause problems if we moved the deleted account to a history file. This would result in two people having the same account number. Our apparent solution appears to be creating more problems than it is solving. Perhaps a much better and certainly simpler solution is to increase the size of the account number. Since our file will allow for almost a billion customers, I don't think we will have a problem and we need not worry about recapturing deleted account numbers. As you can tell, making this decision requires some thought but in the long run the effort will pay off in fewer problems and less work.

Speaking of computer limits, Jerry Weinberg, our teacher in Binghamton, asked us to write a program to solve the traveling salesman problem. Starting at a home point, visiting one hundred places and returning home, we had to arrive at the route that would involve the shortest total distance covered.

My project mates and I figured we'd use the computer to look at all possible combinations, calculate the distances between points, two at a time – using geometric formulas – add them and arrive at the solution. It sounds like a murderous job, but we weren't going to do it – the computer was. We'd start with one hundred random points on a graph, using A0 for the home point and A1, A2, A3, etc. up to A100 for the places to cover. The distance between A0 (x_0, y_0) and A1 (x_1, y_1) is the square root of the sum of $(x_1-x_0)^2 + (y_1-y_0)^2$. If you played hooky the day, we were taught that in math class, trust me. A similar calculation had to be done for all the remaining combinations.

For the number of combinations, just consider traveling to five places. There are five choices for the first location, four left for the second, then three, two and of course, one left. Using the asterisk to indicate multiplication, this becomes 5*4*3*2*1, which equals 120 possibilities. If we have one hundred places but consider only twenty, we have one hundred choices for the first location, then ninety-nine, ninety-eight on down, or 100*99*98*97*96*95*94*93*92*91*90*89*88*87*86*85*84*83*82*81. I'm not going to waste my time calculating that number, but instead use an approximation of 100^{20}, which is larger than the value worked out, but since we still have eighty more locations, it should suffice, actually being a low estimate. The number, 100^{20} equals 10^{40}, since 100 equals 10^2. Don't tell me you cut math again.

According to a 2008 New York Times story, a super computer could perform 1.026 quadrillion calculations per second. This number is approximately 10^{15}. In a year, there are 60*60*24*365, or 31,536,000 seconds. For a thousand years we have 31,536,000,000 seconds. Multiplying these two numbers (approximately 10^{15} calculations per second times about 10^{12}) would result in fewer than 10^{27} calculations for a thousand years. Compare this to 10^{40}. Since we're only considering a fraction of the one hundred locations, we're going to come up way short – by a long shot. We'll need a faster computer – much, much faster. I think Jerry tricked us.

Indeed, a computer can't do everything. It has limitations. This effects business applications as well. Assume that a business needs to shut down the online system for a short time to do some updating of files, say a half hour. If that process takes twenty minutes, that's cutting it close. In today's crazy, stressed out environment of 24/7, how can the corporation even afford to shut down for fifteen minutes? Perhaps the solution is to handle the updates without any shutdown. It will have to be figured out.

Speaking of solutions, specifically the traveling salesman problem, what if I hire ten go-getters to cover the one hundred locations, ten each. I implore that to really hustle for mucho dollars. I turn the dough over to my boss and he is overwhelmed and pleased, completely forgetting about the problem originally assigned.

15. Adding records and calling a program

Now it's time to discuss that program which I asked you to write a few pages ago. It will illustrate adding records to the account number file as well as calling a program to obtain a new account number. The calling program will have the operator enter last name, first name and middle initial. Once all three are entered and valid, an account number will be generated by the system, through the called program,

nextnumb,

which passes back the new account number to the calling program. The called program will read a new file called the next number file, which is defined with the key of account number and set up by the data base analyst. It consists of an indicator, which will be a blank or *d*, and the next available account number. Besides the next account number, there will also be records with account numbers that are available due to the deletion of records from the account number file.

Before any records exist on the account file, the next number file will have one record and the key or next available account number will be 10. If there are yet to be deletions to the account number file and the file has twenty records, the next number file will have one record with the key of 30, with the indicator equal to a space. At this point if someone deleted the record with account number equal to 15, the next number file would have two records, looking like this:

Key	indicator
000000015	d
000000030	

The called program giving us the next account number will be included in the next chapter. The procedure to add an account will be accomplished when the three data fields are correctly entered. To exit this program, the user should enter *xx* for the last name. The lines of the calling program follow.

```
program-name: acctcall
define  main-heading structure
        print-month character(2)
        field character value "/"
        print-day character(2)
        field character value "/"
        print-year character(42)
        field character(74) value "New account number addition report"
        field character(5) value "Page"
        page-number integer(3)
define  sub-heading structure
        field character(54) value "account #    last name          first name"
        field character(57) value "mi "
define  print-line structure
```

```
        field character(18) value spaces
        print-account-number integer(9)
        field character(17) value spaces
        print-last-name character(18)
        field character(17) value spaces
        print-first-name character(15)
        field character(17) value spaces
        print-middle-initial character
define  total-line structure
        field character(43) value spaces
        field character(42) value "The number of account records added was"
        print-count integer(3) mask(zz9)
define  file acctfile record account-record status acct-status key account-number structure
        account-number integer(9)
        last-name character(18)
        first-name character(15)
        middle-initial character
        field1 character(42)
        zip-code integer(5)
        balance signed decimal(6.2)
define data-fields structure
        new-account-number integer(9) value 0
        process-sw integer value 0
define error-msg character(60) value spaces
define in-last-name character(18)
define in-first-name character(15)
define in-middle-initial character
define name-string structure
        field character(26) value "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        field character(30) value "abcdefghijklmnopqrstuvwxyz '-."
define work-date character(8)
define record-counter integer(5) value 0
define page-counter integer(3) value 0
define line-counter integer(2) value 54
work-date = date
print-month = work-date(5:2)
print-day = work-date(7:2)
print-year = work-date(3:2)
screen erase
screen(1,23) "new account number addition program"
screen(5,20) "account number:"
screen(10,20) "last name:"
screen(15,20) "first name:"
screen(20,20) "middle initial:"
```

```
screen(22,36) “enter xx for last name to exit”
input-data: valid-input = 0
        perform input-last-name until valid-input = 1
        valid-input = 0
        perform input-first-name until valid-input = 1
        valid-input = 0
        perform input-middle-initial until valid-input = 1
        perform call-process
        go to input-data
input-last-name: screen(10,36) input in-last-name
        screen(24,1) erase
        if in-last-name = “xx”
                go to end-program
        end-if
        if index(in-last-name, name-string) = 1
                valid-input = 1
        else
                screen(24,20) “invalid characters in the last name – try again”
        end-if
input-first-name: screen(15,36) input in-first-name
        screen(24,1) erase
        if index(in-first-name, name-string) = 1
                valid-input = 1
        else
                screen(24,20) “invalid characters in the first name – try again”
        end-if
input-middle-initial: screen(20,36) input in-middle-initial
        screen(24,1) erase
        if in-middle-initial = space or (>= “A” and <= “Z”) or (>= “a” and <= “z”)
                valid-input = 1
        else
                screen(24,20) “invalid character in the middle initial – try again”
        end-if
call-process: call nextnumb using data-fields
        if process-sw = 0
                account-number = new-account-number
                last-name = in-last-name
                first-name = in-first-name
                middle-initial = in-middle-initial
                field1 = spaces
                zip-code = 0
                balance = 0
                perform write-record
        else
```

```
                if process-sw = 3
                        screen(24,1) "next number file is busy – try again later"
                else
                        error-msg = "next number file error; program ending – press enter"
                        go to end-program
                end-if
write-record: write acctfile
        if acct-status = 0
                record-counter = record-counter + 1
                line-counter = line-counter + 1
                if line-counter > 54
                        perform print-headings
                end-if
                print print-line
                screen(10,36) last-name
                screen(15,36) first-name
                screen(20,36) middle-initial
                screen(5,36) account-number
        else
                error-msg = "account number file problem; program – press enter"
                go to end-program
        end-if
print-headings: page-counter = page-counter + 1
        page-number = page-counter
        line-counter = 5
        print page main-heading
        print skip(2) sub-heading
        print skip
end-program: print-count = record-counter
        print skip(2) total-line
        print skip error-msg
        screen(24,20) error-msg input
        end
```

The program is quite long but only a few lines should be unfamiliar. Note that the report will only display four fields, so we spaced it out across the page. We spread out the fields on the screen rather than bunching them all at the top. Whoever is inputting data will only key in last name, first name and the middle initial. Obviously, we have to check that they don't key garbage so we use the same procedures for inputting the first two fields, which has a few checks. For verifying that the middle initial is valid, we use the line

```
        if in-middle-initial = space or (>= "A" and <= "Z") or (>= "a" and <= "z"),
```

which is a compound logical statement. It just checks to see if what was entered was either a blank or a letter of the alphabet, upper or lower case. The check

(>= “A” and <= “Z”)

will be satisfied if any of the letters from *A* to *Z* are entered and the second part of that check

(>= “a” and <= “z”),

you should be able to figure out. Both sets of parentheses are needed here because we want the input to be between *A* and *Z*, inclusive, without which special characters could sneak in. If you’re wondering why we allow lower case letters of the alphabet for the middle initial, we have to consider people like e. e. cummings. Note that we won’t accept the dollar sign for the middle initial – that’s probably a good thing. The operator can exit the program by entering *xx* for the last name – the only way out. The fields

data-fields

are used to obtain data from the called program, specifically the new account number and

process-sw,

which indicates whether or not the call was a success. The two fields are passed to

nextnumb

with each being 0, since all the information will be obtained from the called program. New keywords are

call

and

using,

both used in the statement to get the data we need. The

call

keyword transfers control from the calling program (the program we’re in) to

nextnumb

and it will use the field

data-fields

to accomplish that. Once the called program gets the next account number, it passes it back to the calling program and the next statement after the call statement is executed. Now we have to make sure the call worked, which it should. The next number file could be busy, indicated by the number 3 in

process-sw.

If that happens, we can still proceed and try again, hoping the file will soon be freed up soon for our use. If the value returned is 0, the call was a success and we have a new account number, so we display it on the screen, write out a new account number file record, produce a line of our report and now another last name can be entered. The update shouldn’t have a problem since the system is generating the account number and the status should be 0.

There shouldn’t be a problem with the call statement, but if there are problems with either the call or the write of the account number file, an error message will be displayed on the screen and printed on the report, with the program terminating. A line will also be printing indicating how many records were written. Those will be for records added to the file even if there are serious problems.

16. The called program and using

As already mentioned, the called program will have a new file called the next number file, which is defined with the key of account number and set up by the data base analyst. It consists of an indicator, which will be a blank or *d*, and the next available account number. Besides the next account number, there will also be records with account numbers that are available due to the deletion of records from the account number file. Once an account number is ready to be sent to the calling program, the next number file will be updated.

Even though the next number file is a keyed file, we will read it sequentially so we always read the first record on the file, taking the first available number and using it until the next number file eventually winds up with one record again. If there are many deletions, this may take a long time but at some point we should wind up with that single record. In any event, our access to get a new account number will work. In the previous chapter, I described what the called program will be doing, so I won't repeat it here. The called program follows.

```
program-name: nextnumb link data-fields
define data-fields structure
        new-account-number integer(9) value 0
        process-sw integer value 0
define file numbfile record next-record status numb-status key account-number structure
        account-number integer(9)
        indicator character
account-number = 0
readnext numbfile update
if numb-status = 0
        perform update-number
else
        if numb-status = 3
                process-sw = 3
        else
                process-sw = 9
        end-if
end-if
go to end-program
update-number: process-sw = 0
        new-account-number = account-number
        delete numbfile
        if numb-status > 0
                process-sw = 9
        else
                if indicator = space
                        account-number = account-number + 1
```

```
                    indicator = space
                    write numbfile
                    if numb-status > 0
                            process-sw = 9
                    end-if
            end-if
    end-if
end-program: end
```

Notice that the program name is

nextnumb,

which appears to be a relatively simple program – for a change. The line

program-name: nextnumb **link** data-fields

has a new keyword,

link,

which enables the passing of data from the calling to the called program. We used the same data name, but they could be different variables as long as the size and characteristics are the same. When the calling program transferred control here, the field

data-fields

had a value of 0 for each field.

data-fields

will be used to pass new values for those fields back to whatever program calls it. Next we define the records on the next number file. It has only two fields, the account number and the indicator. Just as the calling program transferred data, as trivial as it was, we need to pass back data to that program, namely the new account number and a record status. We use

```
define data-fields structure
        new-account-number integer(9) value 0
        process-sw integer value 0.
```

The field

process-sw

will be 0 for a successful process of the next number file. This entire process involves reading the first record there, moving the new account number to the field

new-account-number,

deleting the record just read and adding a new one, unless the account number moved had been used before. If the next number file is busy, 3 will be returned in

process-sw

and if there's a problem with the read, write or delete, a 9 will be passed back. In this case the new account number returned to the calling program will not be used. The new keyword

delete

will only delete the current record, the one we just read with the update to lock it out from anyone else. That's the very first record on the file. It won't delete the whole file, which would really be asking for trouble.

The rest of the program should be familiar but a few comments are in order. We want the first record on the next number file, so we do a sequential read of the first record whose key is larger than 0. That has to be the first record since any account numbers that were used before would appear on the file before the record with a space in the indicator. In the file, all the records are in ascending order in the file.

account-number

was obtained from the read of the next number file because of the

structure

keyword and we'll use it to make sure we delete the correct record.

For any status we return in the field

process-sw,

other than 0, no error message will be displayed on the screen. That will happen in the calling program since we want people to see it. The updating depends on the indicator. If the indicator is *d*, we'll merely delete the record. Otherwise we will delete the record and write another after incrementing the account number by 1. The latter situation means we actually read the last record in the file and so we need to delete it and add another record, which will now be the new last record.

I mentioned a new keyword that allows for a short pause, so let's check it out. The statements

```
        if inquiry-number = 9
                screen(24,20) "Problem with the next number file – program ending"
                pause 10
                go to end-program
        else
end-program: end
```

use the new keyword

pause,

which will delay so that the person at the screen can read the error message. The number after the

pause

represents the time in seconds. The line

pause 1

will only give a break of one second and that may not be enough. Of course, ten seconds may not be significant either if the operator is daydreaming.

Suppose that we didn't have this keyword. We could put logic into our program to accomplish the same result. Assuming that our computer does one million operations per second, the following code would achieve the same delay as before:

```
define pause-counter integer(8)
        pause-counter = 0
        perform pause-loop until pause-counter = 10000000
```

where the procedure

```
        pause-loop
```

would be

```
        pause-loop: pause-counter = pause-counter + 1
```

which involves ten million additions. You could put these few lines in your program and then time the delay. If it still wasn't enough or if it was longer than 10 seconds, you could change it accordingly by modifying the number of the additions. Another possibility is to add the keyword

blink,

which results in the message blinking.

Any of these choices has the possibility of problems – mostly the person keying data not paying attention or falling asleep from too big a lunch. People from outside organizations may be upset if somehow an electric shock is set up to be transmitted to the person keying data, so that's not an option. If the error message is being sent from a called program, it's a better idea to just transmit some value for a variable – as we did here – to eventually get some error message on the screen of the calling program. As you can guess, I'm not about to use the

pause

keyword since the approach of clearing the error message after the data is entered through the

input

keyword is a better idea.

Up until now, instruction was provided to accomplish certain results. Heuristic learning is another method of education that involves learning by making mistakes. Doing things the proper way may be fine until someone flounders and does it incorrectly. Had he done it wrong to begin with, he would only have made the mistake once. A computer can utilize this method to program the game of chess. I'm glad I don't have to do it, but each series of moves – by both the computer and its opponent – is filed and when a move results in a defeat of the computer, the exact sequence is noted and the losing move is never repeated. This happens for every game. You can see that *eventually*, the computer won't lose. Please note the word in italics – it might take a long, long time.

The way chess and Jeopardy are computer programmed is done by methods that involve specific strategies. This means the system will work in the programmer's lifetime, even if the computer loses every so often. What this boils down to is when you play against the computer, you're rally tangling with the person who designed the strategy. Naturally, that person has the advantage of numerous calculations done for him by a machine that a human just couldn't manage so quickly. The strategies involve looking ahead a few moves and proceeding from there.

I programmed two mind games – you may have heard of both – but I'm not sure what language either was in. That's not important. The first was JOTTO, a word game

where you have to guess your opponent's five-letter word before he guesses yours. This is done by offering other five-letter words with a response of from 0 to 5, depending on the jots, or common letters in the secret word and the word guessed. If no letters are in common, the response is 0. If one is, the response is 1. For multiple occurrences of a letter, it's a one-to-one match. For example, if the secret word happens to be *eerie*, guesses of *means* and *level* would result in 1 and 2 jots respectively. JOTTO uses a dictionary of about 1800 words. I had two separate games, one in which you guessed the computer's word and the other had the machine guessing yours – no cheating, please. The former wasn't too exciting but the other is the one I'll concentrate on here. In that approach, the computer could usually figure out the player's word in eight guesses or less.

My strategy aimed at reducing the size of matrix of words as quickly as possible by eliminating any word that couldn't possibly be the secret word. If the computer guessed a word with no letters in common, any word having any of those five letters would be discarded. If 3 was the response, any word without all three of those letters word would be tossed away as well. At the beginning of guessing, I had four words that the computer could choose from – all having different letters – hoping for a few 0 replies. This would really shrink the matrix of words. As you can imagine, it was a great strategy.

The other game was Score Four, three-dimensional Tic-Tac-Toe – almost. The more common game is 3 by 3, but this is 4 by 4 by 4, except that either player only had a maximum of 16 moves at any given time. There were 64 beads with holes so that they could be plopped down on 16 different thin poles, always dropping down as far as possible. Each pole held 4 beads and you could win the game by 4 in a row (length, width or height) or diagonally. The latter had 20 possibilities – you had to be on your guard. The strategy I used for the computer was 1) look for any move that produced a win and if there, end the game 2) look for any move that needed to be made to avoid losing on the opponent's next move and take action and 3) make some offensive move. The program was quite good at suddenly coming up with victory. It didn't lose very often. In either game, I avoided heuristic strategies.

17. Fuzzy math

You might think that the way government balances the budget – yeah, right – is an example of fuzzy math. Maybe that's true, but not what I had in mind here. A few years ago when I was a computer programmer, I got a request from a user for a report dealing with percentages. The specifics of the report are a blur since it was some time ago but the assignment created some difficulties that were not easily resolved. I can relate that occasion by a very pertinent report today. Suppose I want a report on usage on web sites, along with percentages. The access to a site will result in a hit and assuming the sites are Chris, Kelly, Pat, Rene, Whitney and Jamie, the report might look like the following:

WEB SITE ACCESS REPORT

site name	hits	percentage
Chris	0	0%
Kelly	0	0%
Pat	1	33%
Rene	0	0%
Whitney	2	67%
Jamie	0	0%
totals	3	100%

The report looks simple enough but suppose our computer did not round results as we see here. The 67% percentage for the Whitney site would then be only 66% and the sum of the percentages would be 99% and not 100%. Assuming we do take into consideration rounding – either we provide for it in our program or the computer handles it – suppose a week later we ran the report and got the following:

WEB SITE ACCESS REPORT

site name	hits	percentage
Chris	2	22%
Kelly	1	11%
Pat	1	11%
Rene	1	11%
Whitney	3	33%
Jamie	1	11%
totals	9	99%

A month later we got this report:

WEB SITE ACCESS REPORT

site name	hits	percentage
Chris	1	17%
Kelly	1	17%
Pat	1	17%
Rene	1	17%
Whitney	1	17%
Jamie	1	17%
totals	6	102%

What happened in these last two reports was exactly what occurred to me when I created the report I referred to earlier. Obviously the problem has to do with rounding and I explained that limitation to the person who requested this report. That was the first possible solution to the discrepancy but the user wouldn't buy it. What would you do under these circumstances?

From the last two reports you can see that there doesn't seem to be a lot that can be done. The user wanted that total percentage to always be 100%, which was obtained by adding down the column. In the last report, with rounding the number became 102%, but without rounding we get 96%. What I did was to simply change that number at the bottom right so that it was never anything but 100%. I'm not sure if that was accepted or when someone challenged that perfect number I answered that they probably added wrong. I just know that this problem has to occur from time to time – as clearly illustrated by my bogus web reports – and outside of my second solution, there is not very much that can be done about it.

This scenario results because of the math of a computer, which is a bit different from the math that we are used to, unless we were weaned on calculators. If you think that this is something, prepare yourself for some more shocks. Consider the following variables and statements:

```
define w decimal(3)
define x decimal(3.1)
define y decimal(3.1)
define z decimal(3.1)
x = 1/3
y = 1/3
z = 1/3
w = x + y + z
```

The result for the variable

```
w
```

will be 0 if no rounding occurs and .9 if we define the field

```
w
```

as

```
decimal(3.1).
```

We expect that the result should be 1 from adding 1/3 + 1/3 + 1/3 – that's what our math teacher should have taught us. Just the fact that we have defined the first three variables as

```
decimal(3.1)
```

will result in truncation since each will have a value of .3 after the first three assign statements. What happens if we change each of the four variables to

```
decimal(3.2)?
```

In this case will each of the first three variables will be .33 after the first three assign lines and now w will be .99, so we're getting closer to 1. We will get even closer to 1 by

adding more decimal digits but we will never get there unless we somehow round the results. As we have seen earlier, even that won't give us the 1 we desire.

Our expression of 1/3 is equal to the infinite repeating decimal .333333333333…, which I can't show here no matter how many pages of paper I have. The 3s just keep on going, just like the Energizer bunny. On the other hand the computer will give us .3, .33, .333 or something approaching 1/3 but never exactly that. Computer math is finite but we are talking about the infinite and so there will be differences and confusion at times. You may have written code in any language and expect certain results but get something a bit different. The reason could be because of the way a computer does calculations and its limits.

Consider the following statements:

```
define a decimal(3.1)
define b decimal(3.2)
define c decimal(3.3)
define u decimal(3.1)
define v decimal(3.1)
define w decimal(3.2)
define x decimal(3.2)
define y decimal(3.3)
define z decimal(3.3)
u = 1/3
v = 5/8
a = u * v
w = 1/3
x = 5/8
b = w * x
y = 1/3
z = 5/8
c = y * z
```

The symbol used for the first two variables above and a few others represents division. Recalling that the asterisk represents multiplication, what will be the values of

a,
b

and

c?

In the calculation of the first of those variables, note that

u

will be .3 and

v

will be .6, because of truncation. Thus

a

will be .1, and .2 if rounding takes place. The more precise answer is about .208 so we are a bit off. Now using 2 decimal digits,

w

will be .33 and

x

will be .62. The result for

b

will be .20 whether we use rounding or not. It doesn't look like that extra decimal digit made too much of a difference. Using 3 decimal digits results in

y

being .333 and

z

being .625. Now this results in the variable

c

having the value of .208. This will come about with or without rounding. As you can see, that is about the result we wanted and it is a great deal more accurate than the .1 we arrived at initially.

Suppose we had the following:

```
define c decimal(3.3)
define y decimal(3.3)
define z decimal(3.3)
y = 1/3
z = 3/4
c = y * z
```

and if we didn't skip the class on multiplying fractions, we expect a result of exactly .250 for the last calculation. To start with we will have .333
for

y

and .750 for

z.

The value of

c

results in the value of .249 if no rounding occurs and .250 if it does. You can see that using more decimal digits gets us closer to the result we expect from normal math.

As we have seen, we may have differences even if we round results and in some cases that may even be the reason for the unexpected. Not rounding may still create a problem as we have seen in our sample reports but the important thing to realize is why there is a difference. Your job is to explain that discrepancy as well as minimize the damage. That simply means that you may have to allow more decimal digits than you originally had. This is probably more of a warning than anything else. You will need to realize than sometimes no matter what you do, there is nothing further to be done.

18. Deleting accounts

This next program is a relatively easy one. It's time for the program to delete records on the account file. We could incorporate this function into the inquiry / update program but instead we will create a new program. The reason has to do with the fact that we probably only want certain people to have the ability to delete records. Another advantage is that the process will be somewhat easier if deletions are separate. Once we have this program done, we can copy it and modify it and use the result for deleting records on the zip code file. There is no sense writing a separate program when we have most of the logic in place.

The program will allow input of an account number and our program will delete the record, provided the balance is zero. It will then proceed to add the account number to the next number file. As I mentioned earlier, we may not want to use the account number over again – at least for a while – so we would just delete the account record. Adding the deleted record to the other file is just being done for the exercise. A report will be produced to list all deleted accounts. As before, in case of a serious file access, an error message will be printed on bottom of the report and listed on the screen.

Our program will be the following:

```
program-name: acctdel
define  main-heading structure
        print-month character(2)
        field character value "/"
        print-day character(2)
        field character value "/"
        print-year character(43)
        field character(73) value "Account number deletion report"
        field character(5) value "Page"
        page-number integer(3)
define  sub-heading structure
        field character(54) value "account #    last name          first name"
        field character(57) value "mi   street address          city          state"
        field character(19) value "zip       balance"
define  print-line structure
        print-account-number integer(9)
        field character(4) value spaces
        print-last-name character(22)
        print-first-name character(19)
        print-middle-initial character(5)
        print-street-address character(27)
        print-city character(19)
        print-state character(6)
        print-zip-code integer(5)
        field character(2) value spaces
```

```
        print-balance signed decimal(6.2) mask($$$$,$$9.99-)
define  total-line structure
        field character(43) value spaces
        field character(42) value "The number of account records deleted was"
        print-count integer(3) mask(zz9)
define  file zipfile record zip-code-record status zip-status key zip structure
        zip integer(5)
        city character(15)
        state character(2)
define  file acctfile record account-record status acct-status key account-number structure
        account-number integer(9)
        last-name character(18)
        first-name character(15)
        middle-initial character
        street-address character(25)
        field character(17)
        zip-code integer(5)
        balance signed decimal(6.2)
define  file numbfile record next-number-record status numb-status key next-number structure
        next-number integer(9)
        indicator character
define error-msg character(60) value spaces
define work-date character(8)
define record-counter integer(5) value 0
define page-counter integer(3) value 0
define line-counter integer(2) value 54
work-date = date
print-month = work-date(5:2)
print-day = work-date(7:2)
print-year = work-date(3:2)
screen erase
screen(1,30) "account number delete"
screen(4,20) "account number:"
screen(22,20) "enter 0 for the account number to exit"
input-number: input(4,36) account-number
        screen(24,1) erase
        if account-number = 0
                go to end-program
        end-if
        read acctfile update
        if acct-status = 0
                if balance = 0
                        perform process-delete
                else
```

```
                    screen(24,20) “balance is not 0 - not deleted”
                end-if
        else
                if acct-status = 5
                    screen(24,20) “account number “account-number “ is not on the file”
                else
                    error-msg = “account file problem; program ending – press enter”
                    go to end-program
                end-if
        end-if
        go to input-number
process-delete: delete acctfile
        if acct-status = 3
                screen(24,20) “that record is unavailable”
        else
                if acct-status > 0
                    error-msg = “problem updating file; program ending – press enter”
                    go to end-program
                else
                    perform remaining-process
                end-if
        end-if
remaining-process: print-account-number = account-number
        print-last-name = last-name
        print-first-name = first-name
        print-middle-initial = middle-initial
        print-street-address = street-address
        perform get-city-state
        print-city = city
        print-state = state
        print-zip-code = zip-code
        print-balance = balance
        line-counter = line-counter + 1
        if line-counter > 54
                perform print-headings
        end-if
        print print-line
        indicator = “d”
        next-number = account-number
        write numbfile
        if numb-status > 0
                screen(24,20) account-number “ was not added to the next number file”
        end-if
        record-counter = record-counter + 1
```

```
get-city-state: city = spaces
        state = spaces
        zip = zip-code
        read zipfile
        if zip-status = 5
                screen(24,20) "zip code " zip " is not on the file."
        else
                if zip-status > 0
                        error-msg = "zip file problem; program ending – press enter"
                        go to end-program
                end-if
        end-if
print-headings: page-counter = page-counter + 1
        page-number = page-counter
        line-counter = 5
        print page main-heading
        print skip(2) sub-heading
        print skip
end-program: print-count = record-counter
        print skip(2) total-line
        print skip error-msg
        screen(24,20) error-msg input
        end
```

You may be asking why the program is so long if this is a simple process. If we had displayed data on the screen, it would have been even longer. The task is not complicated, but achieving it involves a great deal of logic. We're producing a report of what's happening on paper and access to the files is rather complex. A successful delete means we read the account file, read the zip code file for the city and state, deleted a record from it as long as the balance is 0 and added a record to the next number file. This could be simplified somewhat if we changed the report and only printed the account number, last name and first name, as well as not update the next number file. We chose to print out all the information on the file to help the users.

Note that for the most part, we used lines of code from other programs, such as the update program and the report program. Again, why repeat what has already been done. The main logic is to get an account number to delete and read the account file to make sure that it's there, but also that the balance on the record is zero. It's probably not a good idea to close an open account, so that's why we need this further restriction. If the record can be deleted, we delete it and add an appropriate record to the next number file, with an indicator of *d*. We also print the record that has been deleted on a report with all the fields on the record. Since the account file has only the zip code and not city and state, we have to get those two fields from the zip code file. This results in the structure of the file appearing to be different from before, but it's still the same file.

All along the way as we read, write and delete records, we need to check the statuses of the access to the files. We don't anticipate problems but files can mysteriously disappear and the status checks we perform prevent the program from abruptly ending. If somehow the program does end, at least we'll have a message telling us why that happened. Unfortunately it won't spell out who is the culprit behind the problem.

Many of the procedures are very familiar such as

input-number,

print-headings

and

get-city-state.

They have been changed slightly but the process is similar to earlier versions of these routines. The new procedures are

process-delete

and

remaining-process.

The former deletes the account record – similar to a keyed read or write – and checks the file status after the delete and the procedure

remaining-process

gets the report line ready and prints it as well as taking care of adding a record with the deleted account number to the next number file so that it can be used again. We changed the

end-program

procedure to print the count of the number of records that have been deleted. In addition, if there is a serious file problem, an error message will be displayed at the bottom of the screen as well as on the report.

19. Common statements

In the previous chapter I mentioned that we could finish the delete program for the account number file and then copy it and do a few modifications in order to get a zip code delete program. Obviously the latter would be a great deal less complex since we need not worry about the next number file, although we may want to create a report for the deleted records. At the same time, in producing the account file delete program we stole lines from other programs since the logic was the same for the most part.

There is just one concern in our delete zip code program. We must not delete a zip code record that is used by any account in our system. If we did wipe out that record, it would cause read errors on the zip code file, which we certainly don't want. Thus any zip code must be thoroughly researched to verify that it is obsolete and not still being used by the system. We trust that the people who give us these zip codes to delete are on the ball. The deleting of records that shouldn't have been erased is why there may have been difficulties – which I touched upon earlier – when we shouldn't have problems during access after having done a successful one.

As you go from program to program there will be many similarities as well as identical lines of statements. Just think of the structure for the account record and I think you will agree that it will not change from one program to the next. If it did it meant that something was drastically awry or else we had to accommodate some new field that was added. We want the same structure in every program since we are using the same file in these different programs. When we happen to change the layout of the file, we also want to make sure that we change it in all the programs that use it. We can help ourselves out in many cases by using what is referred to as a copy member. This is some member of another library that consists of lines of code that we are copying into our program. Think of it as something that doesn't change from program to program, at least not very often.

We create the copy member once and simply include it in the program where it is to occur by using the keyword

copy.

Before, the beginning of our program was

```
define  account-record structure
            account-number integer(9)
            last-name character(18)
            first-name character(15)
            middle-initial character
            street-address character(25)
            zip-code integer(5)
            balance signed decimal(6.2)
```

and with the copy member it becomes

```
copy acctmemb
```

where

acctmemb

is nothing more than the previous eight statements above, beginning with the

define

and this saves us some work. In this case, we have brought lines into our program from another source, namely the copy member

acctmemb.

We need that member in some library of copy code from which we can extract it when we run or compile our program. The needed lines will be present when we run our program because they have been extracted from a copy library. We should probably have a separate library for all our copy members and also another library for all our programs. This helps in keeping track of where everything is. Also, if someone else will be in charge of maintaining the programs and copy members, we can tell him where to find the stuff.

Depending on your system and its naming conventions, you may be able to create a copy library called

PROD.COPYLIB.

Since you also have need for a similar test library, that one might be conveniently called

TEST.COPYLIB.

The name of the two libraries for the computer programs could be simply

PROD.SOURCELIB

and

TEST.SOURCELIB.

You could call these libraries by some other names but these above suggestions are probably easier to remember and more meaningful.

Getting back to the copy member, it has a few advantages. The obvious one is that if we change it, we won't have to change every program that uses it. If we didn't have that copy member and we changed the layout for the account file record, one program would have the change but other programs wouldn't. The result might be that the other programs would abend. Even if they didn't, someone looking at the other program without the modification to the layout might get misinformation about fields in the record since what the program has is out of date.

Using the copy member might require every other program to be recompiled since the program as it was compiled earlier would have had the old layout until we compile it again. However, that is a bit easier than adding three new fields to every program that uses that copy member. In the case of the P language, we won't need to do anything else because the execution of our programs is dynamic. This means that when we run them, the version of the copy member as it currently exists is extracted from the copy library. In our case the process couldn't be easier. For other systems it may not be that simple.

We can use copy members for structures, procedures or for lines in a program that are found from one program to the next, such as counters and heading layouts for reports. This could come in quite handy for some kind of date routine or file access that is used repeatedly. On the other hand, I have seen programs with one copy member after another so that it takes some effort to see what the program is doing. You really have to expand the code to get the entire program in front of you since

copy acctmemb

doesn't really tell you what fields are in the record. That's the disadvantage.

Another disadvantage is that you may have a long list of fields in a record when all you really cared about was one or two. These you could have gotten by simply defining the fields needed and not worrying about the others. The same could apply to a copy member that has a list of statements that are used for some group of routines, like file access. Say the copy member contains all the code for reading, writing, deleting and rewriting records to a file but you only need to read the records. You still would have all the code including lines that you didn't need. This tends to make the program longer and the more statements in a program, the more difficult is it to decipher and maintain.

As an example of a copy member that has executable statements rather than merely the layout of a file record, consider a routine that we use repeatedly to verify that dates are valid. We will assume that all the dates are eight characters and the date should represent a valid date in the format yyyymmdd. We will have to verify that it is numeric. The copy member to do this will consist of the main procedure

check-date,

along with three others. These are the statements you see below.

```
check-date:     if date-work >= "00000000" and <= "99991231"
                        if date-work not = "00000000"
                                perform range-check
                        end-if
                else
                        date-switch = 1
                end-if
range-check:    if date-yyyy > 0
                        if date-mm > 0 and <= 12
                                if date-dd > 0 and <= 31
                                        perform validity-check
                                else
                                        date-switch = 2
                                end-if
                        else
                                date-switch = 3
                        end-if
                else
                        date-switch = 4
                end-if
validity-check: if work-dd > 28
                        if date-mm = 2
                                perform february-check
                        else
                                if work-mm = 4 or 6 or 9 or 11
                                        if work-dd > 30
                                                date-switch = 6
                                        end-if
```

```
                                    end-if
                                end-if
                        end-if
february-check: leap-switch = “n”
        if date-yyyy(3:2) = 0 then
                if mod(date-yyyy, 400) = 0
                        leap-switch = “y”
                end-if
        else
                if mod(date-yyyy(3:2), 4) = 0
                        leap-switch = “y”
                end-if
        end-if
        if work-dd > 29 or leap-switch = “n”
                date-switch = 5
        end-if
```

We also need the variables from the define statement

```
define  date-work structure
        date-yyyy integer(4)
        date-mm integer (2)
        date-dd integer (2)
define date-switch integer
define leap-switch character
```

to make this all work.

To get started we have to set the variable

date-switch

to 0 and

date-work

to the value of the date that we are checking. The next step is to perform

check-date.

Note that the resulting values from 1 to 6 for

date-switch

mean that the date is invalid, with the following meanings:

1 - date not numeric
2 - day out of range
3 - month out of range
4 - 0 is invalid for the year
5 - invalid day for February
6 - that month has only 30 days.

Some programs that check for valid dates stop at the range check and don't pursue validity checks. This means that a date of June 31 is acceptable as well as February 30 but our logic won't accept those as valid dates. Nor will it accept February 29, 2100 since that year will not be a leap year, as far as I know. That is why we have so many statements in the copy member. However, once we write it and test the logic, we can use the same copy member in any program that does date validity checking.

The first check will allow for the date to be all zeroes. Then we have

```
if date-yyyy > 0
```

to check that the zero is entered for the year. Valid years can be 1 and –1, with the latter representing 1 BC. We'll be concerned with starting at the year 1 and going up to December 31, 9999. Someone else can take care of other dates. The next two statements relating to

```
date-mm
```

and

```
date-dd
```

check for valid months and days of the month, with

```
if date-mm > 0 and <= 12
```

looking to see that the month is between 1 and 12, inclusive. The logical operator

```
<=
```

represents less than or equal. You've seen it before. In the range-check procedure the variable

```
date-switch
```

could be 2, 3 or 4 depending on whether the day was out of range, the month was out of range or the year was entered as zero, respectively.

If the date passes these initial checks, we perform the procedure

```
validity-check.
```

This makes sure we don't have dates like April 31 or February 29 if the corresponding year is not a leap year. If the day is 28 or less, we know that this is a valid date and we are done with checking. That is why the first line is

```
validity-check: if work-dd > 28
```

and we start with a check for February. If the month is 2, we do the February check, which I will get to later. Otherwise we check for the 30-day months and if we have one and the day is 31, we have an invalid date. Any other date that is not in February will pass the test and we are done. The procedure

```
february-check
```

simply verifies that a day of 29 corresponds to a leap year. If we don't have a leap year or if the day is greater than 29,

```
date-switch
```

will be set to 5, since the date is invalid. To check for the leap year, we have two possibilities. The first is that the year could end in two zeroes, like 1900 or 2000. That year will be a leap year only if the year is perfectly divisible by 400. Thus 2000 was a leap year but 1900 and 1800 were not. I wasn't around in the latter two cases but take my word for it. If the year doesn't end in two zeros, it will be a leap year if the year is divisible by 4. This brings us to our next keyword,

mod.

The line

mod(date-yyyy, 400)

will be 0 if the variable

date-yyyy

is exactly divisible by 400. If it not, the result of the line will be the remainder of that division, namely the year divided by 400. Thus a result of 0 means that we have a leap year and so the variable

leap-switch

is set to "y". Initially this field had the value of "n" so if we have a positive remainder, nothing further happens and we don't have a leap year. With that in mind you can now figure out what

if mod(date-yyyy(3:2) ,4) = 0

does. To start with it takes the full year but only uses the two rightmost digits and tries to divide by 4. If there is no remainder, we have a leap year. Otherwise we have an invalid date. If you suggest that we could have used the full year instead, you're correct and you're paying attention but we could just divide the last two digits of the year since it won't matter what the two leftmost digits are. That's all there is to verify a date. I think you will agree that this is a useful copy member to have.

20. Arrays

In the previous chapter we talked about copy members for the date routine. Note that we had six different error messages depending on what was wrong with the date in question. Once we get through with the date check, assuming this is an online program, we need to check for any error and if there is one, we need the statements,

```
if date-switch = 1
        screen(24,20) "date not numeric"
end-if
if date-switch = 2
        screen(24,20) "day out of range"
end-if
if date-switch = 3
        screen(24,20) "month out of range"
end-if
if date-switch = 4
        screen(24,20) "zero is invalid for the year"
end-if
if date-switch = 5
        screen(24,20) "invalid day for February"
end-if
if date-switch = 6
        screen(24,20) "that month has only 30 days"
end-if
```

We should use this for every date we check and we could easily make it a procedure. What we would do is move the date to be checked to a very specific new date field and then pass this date to the procedure. Once the routine is done, we would have a value for the field

date-switch,

which would tell us if the date was valid or why it wasn't, depending on one of the six values resulting from bad data. This checking could be put into a copy member but we have another option. We can put each message into an array, with the messages, *date not numeric* corresponding to 1, the message, *day out of range* corresponding to 2 and so forth. With this in mind we won't need to use the above if statements since we will just reference the message corresponding to

date-switch,

which is the variable we use for determining what error has to be displayed. The fields that are necessary can be seen in these statements.

```
define message-array character(180) element t-element character(30)
        occurs 6 times structure
field character(30) value "date not numeric"
```

```
field character(30) value "day out of range"
field character(30) value "month out of range"
field character(30) value "zero is invalid for the year"
field character(30) value "invalid day for February"
field character(30) value "that month has only 30 days"
```

We now can replace all our

```
if
```

statements related to these errors by the three statements

```
if date-switch > 0
        screen(24,20) t-element(date-switch)
end-if
```

with

```
t-element(date-switch)
```

producing one of six message based on the variable in parentheses.

```
date-switch
```

will be between 1 and 6, inclusive, since there are only that many error possibilities. We could even add other messages to the array not related to date checking. If we did, we would have to adjust the size of the array, in this case 180. In addition, the count spelled out by the keyword

```
occurs
```

would have to be larger. Then we can add error messages as we need them with only a few modifications.

In summary, the statements,

```
define message-array character(180) element t-element character(30)
        occurs 6 times structure
field character(30) value "date not numeric"
field character(30) value "day out of range"
field character(30) value "month out of range"
field character(30) value "zero is invalid for the year"
field character(30) value "invalid day for February"
field character(30) value "that month has only 30 days"
```

simply define an area of 180 characters for our array, broken down into 6 pieces or messages, each one being referred to as the variable,

```
t-element,
```

which consists of 30 characters each and we shall refer to these messages by subscripting. The new keywords are

```
element,
occurs
```

and

times.

This is how we configure the array and note that the above statement not only defines but also give our array values. The subscript turns out to be the variable

date-switch

and it must be a numeric field. In this case it is between 1 and 6 inclusive since we have only 7 possible values for this field, 0, 1, 2, 3, 4, 5 or 6 and

```
if date-switch > 0
        screen(24,20) message-array(date-switch)
end-if
```

guarantees that our subscript does fall within this range of 6 positive numbers. A value of 0 indicates no error at all.

Let us return to the index keyword we used earlier. This was needed to verify that the letters in the name fell within a certain range of letters and symbols. It worked rather well but suppose it didn't work as we described and it could only look at one character at a time to see if it was found in a different string. If the variable

alpha-string

represents the set of valid characters for first name mentioned earlier, namely the letters of the alphabet, both lower case and capital letters, as well as certain special symbols, then

index("A", alpha-string)

would result in the value 1 since *A* is certainly in the string but

index("3", alpha-string)

would give a value of 0 since the number *3* is not in the string. With that in mind, we could still do character validity checking using these statements:

```
define i-sub integer(2)
define work-array character(15) element c-element character
        occurs 15 times
define process-sw integer
i-sub = 0
process-sw = 0
work-array = first-name
perform check-string until process-sw > 0
check-string: i-sub = i-sub + 1
        if index(c-element(i-sub), alpha-string) = 0
                process-sw = 2
        end-if
        if i-sub = 15 and process-sw = 0
                process-sw = 1
        end-if
```

What we are doing is going through each character of the string and verifying it against the string of acceptable characters. As soon as we find that a character is not in the string, we can stop the search since we know that the first name keyed is invalid. This is handled by the statement

if index(c-element(i-sub), alpha-string) = 0,

which will end the checking. Note that when

process-sw

is 1, the name meets the requirements but a value of 2 for the variable

process-sw

means that we have encountered a character that is not in the variable

alpha-string.

We do the checking as many as 15 times because first name can have as many as 15 characters. Note that if the name is only 6 characters, the remaining 9 will be spaces but that is a valid character so the test will be satisfied. This procedure is a bit more work that using the keyword

index

as we originally described but it will get us the result that we desire. The statement

perform check-string **until** process-sw > 0

keeps doing the procedure

check-string

until

process-sw

is no longer 0. This is achieved in one of two ways as we discussed previously. Note that

i-sub

starts out as 0 but will be 1 next and then could eventually be 15, but not any more than that.

Now suppose that we didn't have the keyword

index

at all, but still needed to do character validity checking. It could be done by using the the

perform

statement. The statements to do this are

```
define work-array character(15) element c-element character
        occurs 15 times
define c-string character(56) element t-char character value
   "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz .-'"
define i-sub integer(2)
define t-sub integer(2)
define process-sw integer
define first-name character(15)
i-sub = 0
process-sw = 0
work-array = first-name
```

```
perform check-string until process-sw > 0
check-string: i-sub = i-sub + 1
        perform check-character varying t-sub from 1 by 1 until t-sub > 56
        if i-sub = 15 and process-sw = 0
                process-sw = 1
        end-if
check-character: if c-element(i-sub) = t-char(t-sub)
                        t-sub = 57
                 else
                        if t-sub = 56
                                process-sw = 2
                        end-if
                end-if
```

Before looking at the statements, consider what we have to do to validate the characters of the name. We must take one letter at a time starting with the leftmost position and check it against the string of valid symbols. That string is

c-string

and if the character is in the string, we then take the next character and proceed in a similar manner. If we find that each character is in

c-string,

the name passes the validity check. However, if we find any character that is not in

c-string,

we can stop the checking and need go no further since all other parts of the name don't have to be verified. Once again the space is a valid character so it will not result in an invalid name.

With that in mind, the statement

perform check-string **until** process-sw > 0

will result in either a value of 1, which means the name is valid or 2 which indicates that it is not since at least one character is unacceptable. Note that as soon as we get through any character whether it is the first one, the last one or any other and it is not valid, the process will stop. The statements,

```
if t-sub = 56
        process-sw = 2
end-if
```

enable that to happen since this indicates that we are at the very last position in

c-string

and we have yet to get a match.

In the

check-character

paragraph note that

```
if c-element(i-sub) = t-char(t-sub)
        t-sub = 57
```

will stop further checking of a character in the name against

```
c-string
```

when there is a match. At this point we just go on to the next character in name.

The statements,

```
if i-sub = 15 and process-sw = 0
        process-sw = 1
end-if
```

mean that we have gotten through every character in the name and each one is in

```
c-string.
```

Thus we set

```
process-sw
```

to 1 and the name passes the validity check. Had

```
process-sw
```

not been 0, it would have stayed whatever value it was and as you might expect it would have been 2, which indicates the name was invalid.

The last statement that I need to talk about is

```
perform check-character varying t-sub from 1 by 1 until t-sub > 56
```

which is a new variation of the perform keyword. It will start with a value of 1 for

```
t-sub
```

and loop through the paragraph

```
check-character
```

and as it goes through this paragraph it will increment

```
t-sub
```

by 1. In this case the

```
from
```

and the

```
by
```

are the same, but they could be different. This varying process will be done until

```
t-sub
```

is 57, that is, it's greater than 56. This is accomplished by

```
varying t-sub from 1 by 1 until t-sub > 56.
```

You can probably see that it won't always do this perform 56 times since once we get a match, we set

```
t-sub
```

to 57 so that we can go on to the next character of the name.

Every other line of the process should be familiar to you. That is all there is to the validity checking for the name. We could use the same check for last name, street address and city although we might want to use a slightly different set of characters in our string of valid characters depending on what we are verifying. Obviously our keyword

```
index
```

would have been a lot more convenient but sometimes we have to survive on what our system provides us.

21. Down in the dumps

Before I get into *dumps*, let's review what happens when we run a program. The program is somewhere in the computer and each statement or instruction as well as data structures and the data itself could be in the machine. Much of this can be found in an area called *working storage*, and it is nothing more than a temporary place for a program as it gets executed, or run. If there is a problem as the program runs, we will have the *luxury* of looking at these areas of working storage to see what the problem is. As you know, a program that has difficulty running is said to abend and the result is a dump, or more specifically a picture of working storage. But it is not a pretty picture.

A dump shows instructions in our program and data in almost unreadable format. After all, the computer processes programs that we wrote as object code or machine language, which is foreign to us. The result is a dump in hexadecimal format, which happens to be base 16. We will need to be able to count in that base in order to read a dump. Fortunately with all the tools and advances in information technology, we really won't need to worry about reading dumps on a serious level. If you own a PC and sometimes run into unexplainable problems (if you don't have crashes you are probably from another planet), the result on your screen is a meaningless message, or perhaps nothing is happening. A restart of the PC will remedy that.

The early days of computing forced people to know hexadecimal and how to read dumps. I'll talk about specifics regarding other base systems in the next chapter but for now I will summarize what a dump involves. To begin with, you could get a dump if you tried to run a program and that program couldn't be found on the computer. Maybe you put the executable program in library B but the computer was looking in library A so that's why it wasn't found. The solution is either put the program in library A or point to library B when you run the program. If we have the *program not found* scenario we might get a dump along with some kind of system code, which right away indicates to us that the program was not in the library where it should have been. Thus the dump was not really needed.

You could also get a dump if you tried to create a file but didn't allow enough space for the records in the file. Once again you would see a familiar system code along with the redundant and unnecessary dump. After getting the same system code, you would easily recognize that your file needed more space. A similar situation would occur for other little problems and in most cases, the system code would tell you all you needed to know without any need for that hexadecimal junk.

However, back in the old days there may have been other times when you had bad data and you got a dump along with a certain system code. You might recognize that the system code indicates bad data but it wouldn't tell you which record and what field caused the problem. On that occasion you had to dig into the dreaded dump. There was a specific code that warned you of an attempt to divide by zero, which is not allowed since it can't be done. If you ran into this system code, you could just search your program for the division operation since it was rare in programs and you would have a fairly good idea of your problem. Bad data other than that was another story altogether.

You could actually figure out what was wrong without reading the dump in many cases. Of course you would need to know at about what statement your program abended. If you could figure that it was one of two or three lines, you could see where the difficulty lay. The statement that was the problem was more or less spelled out to you as an interrupt, that is, there would be some statement to the effect

INTERRUPT AT 003A6

but you had to interpret which line that represented in your program. The way to do this involves looking at the listing of your program, which has numbers somewhere relating this number to some line of your program. That is to say, this strange number

003A6

actually points to the statement in your program where the interrupt or problem occurred. The listing that correlates your program statements to another set of numbers is called a PMAP, which stands for procedure map and it ties the statement number to a number in storage.

Of course you could look at the PMAP and not find the

003A6

anywhere but you saw on two successive lines

003A2

003B4

and this would indicate that the statement corresponding to

003A6

was either of the two lines above. This is so because our strange number is a hexadecinmal number between 003A2 and 003B4. I don't expect you to understand that since you probably don't know how to count in base 16 so take it on faith for now. Since our interrupt is between these two numbers, this means that one of the statements corresponding to these two numbers is what caused our program to abend.

The complicated procedure above is how I used to track down abends in programs, provided I had a PMAP. I would then look at the statement or two and since only a variable or two was involved, I quickly got to the root of the problem. In addition, the dump would actually show you what was in the variables but you had to find that area of working storage in the dump and it may not have always been easy.

Today we don't actually need to worry about this technique as some software tool might be very specific in pointing out not only the statement where the problem exists but also the field that's in error. All you have to do is look at it and proceed from there. If the variable in question wasn't specifically spelled out, you could look at the troublesome statement and check each variable on that line. If a variable was supposed to be numeric you might easily discover that it had spaces in it and that was the cause of the abend.

Whenever I worked with testing programs, I always felt there were only two possibilities. Either I wrote the program from scratch and so I was very familiar with the goings on of the process, or I made a small change to an existing program. If the program was new and abended, I would probably know without too much trouble what was wrong since I had a good grasp of all that was happening. If the program wasn't new and involved a change, there was a high probability that the abend occurred at the line that was modified or something related to the change. That thinking usually worked for me.

There are certain abends that can't be avoided but some should never occur. A zero divide is one and it can be avoided by doing a check before the actual division takes place. If the divisor is zero, don't perform the operation. As far as bad data goes, do as much as possible to minimize these interruptions. Data comes from input or it can be generated by the system. If the latter, that is, if some program generates the data, you should always have good data. If not, make changes to the programs generating the data so that it is always integral.

On the other hand, if the data is keyed into the system, you have less control but you can always do preliminary checking and reject invalid data, especially fields that should be numeric and aren't. Even if the data is coming from an outside source, you can put in similar verifications before the actual processing. This will mean fewer if any abends due to bad data. Not long ago I had just this situation where a file was originating from outside and we had a program that did initial checking to make sure the fields were what they were supposed to be. If they weren't, the next program that actually processed the data just didn't run.

That last action may have been rather drastic but maybe it had to be done. Another alternative might be to skip the record in error and go on to the next good record. This approach will still prevent the possible abend and you'd probably care to put some kind of message on an error report for the record in error. Someone would then see it and take appropriate action.

Other problems, like I/O errors, could result from a tape drive that merely needed cleaning. Space problems and *file not found* can be minimized by diligence in what should be happening. Obviously you can't prevent all problems, but taking care of the majority so that they are almost an impossibility of occurring will keep everyone happier. A bit of thought and analysis ahead of time will make weekends and evenings almost virtually trouble-free.

Reading dumps and determining what each system code means will be a part of your learning process when you begin a new job. Fortunately there should be documentation to aid you in these concerns and your fellow workers will be more than happy to guide you in learning as much as you need to know. If your compatriots aren't helpful, make a batch of brownies with Ex-Lax, remembering not to indulge. Each company will have different problems as well as different tools to solve them and in a matter of time you will be comfortable in these environments.

Let's get back to the method I suggested before for obtaining zip code data. We'll use something similar to an array, called a table. For this new program,

accttable,

we'll steal code from

acctsly

of chapter 13.

```
program-name: accttable
define  file acctfile record account-record status acct-status structure
        account-number integer(9)
        last-name character(18)
        field character(58)
        zip-code integer(5)
        field character(10)
define  table  ziptable  source  prod.copylib(ziptable)  record  zip-record  key  zip  found  t-sub
        structure
        zip integer(5)
        city character(15)
        state character(2)
define error-msg character(60) value spaces
define list-city character(15) value spaces
define list state character(2) value spaces
screen erase
screen(1,23) “account number inquiry with table”
screen(4,20) “account number:”
screen(6,20) “last name:”
screen(8,20) “city:”
screen(10,20) “state:”
screen(12,20) “zip code:”
screen(22,20) “to exit, enter 0 for the account number”
input-number: input(4,36) account-number
        screen(24,1) erase
        if account-number = 0
                go to end-program
        end-if
        read acctfile
        if acct-status = 0
                perform get-city-state
                screen(4,36) account-number
                screen(6,36) last-name
                screen(8,36) list-city
                screen(10,36) list-state
                screen(12,36) zip-code
        else
                if acct-status = 5
                        screen(24,20) “the account number “account-number “ is not on the file”
                else
                        error-msg = “account file read problem; program ending – press enter”
                        go to end-program
                end-if
        end-if
```

```
        go to input-number
get-city-state: search zip-table key = zip-code
        if t-sub = 0
                screen(24,20) "that zip code " zip-code '" is not on the file"
                list-city = spaces
                list-state = spaces
        else
                list-city = city(t-sub)
                list-state = state(t-sub)
        end-if
end-program: screen(24,1) erase screen(24,20) error-msg input
        end
```

You'll notice some similarity to what we had in the arrays earlier. New keywords are

table,

source,

and

found.

In the lines,

```
        define  table ziptable source prod.copylib(ziptable) record zip-record key zip found t-sub
                structure
                zip integer(5)
                city character(15)
                state character(2)
```

the first keyword tells the system a few things:

ziptable

is both a copy member in the production copy library, PROD.COPYLIB (the second occurrence), as well as a table (the first), which enables us to go through it with a search – another keyword. That one,

source

points to the proper library and the right member in it,

ziptable.

The keyword

found

magically places into the field,

t-sub

the number of the row number in the table which matches the zip code from the account number file. This gives us the corresponding city and state for displaying on the screen. If there is no match,

t-sub

will be 0. Note that that variable is not defined but based on the size of the table it will be an appropriately sized integer. This occurs because of the combination of the keywords

search

and

found

in the definition of the table. This all takes place because the table is dynamic, meaning it's current. From the definition of the table, all the values in the copylib member will be available to us. The size of the table results from the copy member.

The statement,

get-city-state: **search** zip-table **key** = zip-code

means we don't have to read the zip code file anymore. There are fewer lines of code in the program and it may run faster that Sly's program, despite his name. You can see that the

search

keyword is quite powerful as it looks for a match and because of the way the table is defined,

t-sub

is the row number of the match, giving us the city and state corresponding to the zip code. There shouldn't be any situation where there isn't a match because of the connection between the zip code table and the zip code file we used before since tech support updates the zip code table directly from that file. We'll display an error message anyway if that does happen.

22. Base systems

Computers use a variety of different number base systems to operate, including the familiar base 10. In addition, binary or base 2, octal or base 8 and hexadecimal, which is base 16, are also used. If you are going to get an appreciation of just how computers work, you will need some comprehension of the different number systems. We will begin by doing some counting in all four of these bases.

We will count in each from 0 to 23 and we shall start with the familiar base 10. If you have a PC, access the calculator on your system by clicking on start and then go up to programs. From there go to accessories and then click on calculator. Once there, click on view and change your calculator to scientific and you should be in base 10 or decimal. If not, set the calculator to that base by turning on

Dec

from the four choices Hex, Dec, Oct, and Bin. If the last choice is Bin Laden, call the FBI. You are now ready to begin counting. If you don't have a PC, you may be able to accomplish the same results with an ordinary calculator by doing the following:

click on 0, then +, then 1 and then =.

The result will be 1. If you then repeat clicking on the = sign, you will get the same result as if we were counting from 0 to 23. The result from your PC's calculator, your ordinary calculator, or if you have neither should be

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23.

This is what we expect to get if we were to count from 0 to 23 in that familiar number base 10.

Now we shall repeat the process for binary or base 2. First set the PC calculator to base 2 by turning on the switch for

Bin.

Proceed as before and the result will be

0 1 10 11 100 101 110 111 1000 1001 1010 1011 1100 1101 1110 1111 10000 10001 10010 10011 10100 10101 10110 10111

and this is counting from 0 to 23 in base 2. Again if you don't have the scientific calculator, take my word for it and I will return to these numbers shortly.

We will now repeat the process for octal and hexadecimal. Switch on base 8 by setting the calculator to

Oct

and repeat the process as before. The result will be

0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 20 21 22 23 24 25 26 27

and that list represents counting from 0 to 23 in base 8. Next set the calculator to base 16 by clicking on

Hex

– don't worry, nothing bad will happen to you – and by once again repeating the exercise, the result is

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17.

This is the sequence of numbers you get if you count from 0 to 23 in base 16 or hexadecimal.

If you are puzzled by this, don't be, as I will attempt to clarify these sequences above. To begin, note that counting in base 10 involves the 10 symbols 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The number 23 in base 10 has 2 and a 3, where the 2 represents 10s and the 3 represents units or 1s. If we consider the number 328 in base 10, the 3 stands for hundreds or 10 x 10, the 2 represents tens and the 8 stands for units. Hundreds, tens and units are all powers of 10 as the hundreds represent 10 to the second power, the ten represents 10 to the first power and unit represents 10 to the zero power. Any positive whole number to the zero power is 1, whether it be 10, 16, 8 or 2. Trust me.

This you may have learned in math when you studied powers of numbers. If you missed that class a quick lesson in factors and powers should convince you of this. Powers of a number are nothing more than the number of times it is used as a factor. Thus

10 x 10 x 10

uses 10 as a factor three times or we could say that this represents

10 to the power of 3,

or

10 to the 3rd power.

If we divide

10 x 10 x 10

or

10 to the 3rd power

by

10 x 10

or

10 to the 2nd power,

the result is 10 since we are dividing 1000 by 100 and we know that the result is simply 10. You may have also learned that you could accomplish the division by merely subtracting the exponents or powers of 10. In this case we would subtract 2 from 3 giving 1 and we would still come up with 10, or 10 to the 1st power.

With that in mind, consider dividing

10 x 10 x 10

by

10 x 10 x 10

or

10 to the 3rd power

by

10 to the 3rd power.

Subtracting the exponents 3 from 3 gives 0 or our answer is

10 to the 0 power.

But we know that 1000 divided by 1000 is equal to 1 which illustrates that

10 to the 0 power

is equal to 1. Didn't I tell you? If we did a similar division using a 6 rather than the 10 you could see that any positive number to the 0 power will be equal to 1. Note however that 0 to the 0 power is not equal to 1!

This brings us back to our number 328, which is

3 x 10 to the 2nd power + 2 x 10 to the 1st power plus 8 x 10 to the 0 power

or

300 + 20 + 8.

This will be the makeup of our number in base 10 and it will apply to any number we can consider in the decimal system. But the same makeup will apply to other bases, whether it is base 2, base 8, base 16 or base 62 for that matter.

Looking at our process of counting from 0 to 23 in the other three bases, you may have noticed the makeup of each of these number systems. Base 2 has only two elements, 0 and 1 while base 8 uses the 8 elements 0, 1, 2, 3, 4, 5, 6 and 7. Finally hexadecimal needs 16 characters so we have to add a few more that are not familiar to us, namely A, B, C, D, E and F. The A would be the equivalent of 10 in base 10, B would stand for 11, C would represent 12 and you can figure out the other 3 symbols. Thus base 16 uses in order the 16 elements

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

If we were to think of a base 62 system, you might realize that we need 62 symbols and we might use the 10 familiars numbers 0 through 9, the capital letters of the alphabet and lower case letters and we would have our 62 elements. We won't get into this system but you can see how different bases systems rely on different combinations of symbols.

Let us now look at three numbers, one in each of our other base systems. These are

10111 (base 2),

27 (base 8)

and

17 (base 16).

These are all equivalent or equal to 23 in base 10.

10111

can be broken down into

1 x 2 to the 4th power +

0 x 2 to the 3rd power +

1 x 2 to the 2nd power +

1 x 2 to the 1st power +

1 x 2 to the power of 0.

This gives us

16 + 0 + 4 + 2 + 1 = 23.

The number in base 8 of 27 can be broken down into

2 x 8 to the 1st power +

7 x 8 to the power of 0.

This translates into

16 + 7 = 23.

Lastly our number in hexadecimal of 17 is equivalent to

1 x 16 to the 1st power +

7 x 16 to power of 0.

This also becomes

16 + 7 = 23.

You can translate back and forth between the different number bases on the scientific PC calculator by entering 23 in base 10 and then switch to binary, octal or hexadecimal. This will work for larger numbers as well.

You may wonder what advantage there is to using base 16 or base 2. Obviously binary involves only two symbols, 0 and 1 which can be translated into off or on in a computer. This represents whether a current in a circuit is flowing or not. The disadvantage to base 2 is that numbers would require more digits to capture the number 23, or just about any other number, for that matter. Note that we could have a larger number in hexadecimal stored with fewer digits than in base 10. So there are advantages and disadvantages of each base system. It may not be obvious but you can readily translate a number from binary to hexadecimal without much trouble. I should say the computer can do that, not us humans.

Though we are more familiar with counting in base 10, we should have a little knowledge about how other base systems work. Fortunately there are scientific calculators that can come to our rescue. With this small treatise, there should be more understanding when we do use the calculator and other base systems. In addition, it now might make a bit more sense as to what all that garbage is that shows up when we get a dump from a computer program. We may not understand it, but we will know that all those numbers represent working storage in hexadecimal or base 16.

Computers can do a lot for us, but can they rule the world? I wouldn't bet the ranch, even if I owned one. As I mentioned in the discussion on heuristic learning, programs are all designed by people, using strategies created by individuals. I know, some human beings rule the world – a few not very well. Others want to rule. I once designed a computer program to create another program that would produce reports. The former required input on the part of the person who wanted the report. Since reports are so different from one another, I wondered if all the extra effort was really worth it.

Many people talk about this computer replacing human beings, but I'm not convinced. When a computer can experience emotions such as joy, love, fear and sorrow – and even hate – I'll be concerned about the robots. A more recent example to back up my feeling can be observed now when you call a corporation, wanting to talk to a person. Instead, you talk to a *smart* computer. Maybe that word in italics is similar to the meaning in our youth when a parent responded to us, "Don't get smart!" I don't think we have any worries for the present, especially when the computer asks us a question requiring only a *yes* or *no* reply. When we reply *no*, the computer retorts, "I don't understand you." Do you think we should have responded with *nyet*? Obviously, the system needs some work.

23. Sorting bubbles

We won't actually be sorting bubbles here, but instead use what is referred to as a bubble sort. One common procedure in computer systems is the sorting of files of data. We might need a report sorted by account number using a particular file and later desire another report from the same file but sorted by a different field, such as last name. Many systems will use the same file but sort it in a different order and produce different reports for different people in an organization. There are also other times when it is advantageous to sort a file before doing any processing.

Fortunately it is not difficult to sort a file of data. In some systems you can merely go into edit mode in the file and enter a simple command and your file will be in a different order. Another possibility is to have some kind of job control language that will result in the file being sorted. We'll get into those methods later. For now suppose we had a file that was in account number order, but needed to be sorted by zip code. Assume also that we didn't have any way of sorting the file but had to rely on some technique that we ourselves developed in a computer program. Simply put, we had to sort the file ourselves.

The way we could do this would be by reading the entire file into a table of data and then use some logic to sort the table. Once done, we would write out the table to a new file, which would be in the sorted order we desired, in this case by zip code. Our report could then come from that new file. Another option would be to take the sorted table of data and process it as the file we really wanted in our desired sort order to produce the report.

Here we'll read the account number file and move the records to a table, sort the table and then create the sorted file from the sorted table. I'll leave it up to you to produce the needed report. Hint: just run the new file through the program

acctlist.

For simplicity sake we shall assume that our file has no more than 100 records, although we don't know the exact count. The process to do all this – minus the processing for the report – is accomplished by the following statements:

```
program-name: bubblesort
define work-array character(10000) element c-element character(100) occurs 100 times
define i-sub integer(3)
define t-sub integer(3)
define process-sw integer
define record-count integer(3)
define file acctfile record account-record status acct-status structure
        field character(100)
define  file newfile record new-record status new-status structure
        field character(100)
define error-msg character(60) value spaces
define hold-line character(100)
process-sw = 0
i-sub = 0
```

```
work-array = spaces
account-number = 9
perform load-array until process-sw > 0
perform sort-array varying i-sub from 1 by 1 until i-sub = record-count - 1
perform write-file varying i-sub from 1 by 1 until i-sub = record-count
go to end-program
load-array: i-sub = i-sub + 1
        readnext acctfile
        if acct-status = 0 then
                c-element (i-sub) = account-record
        else
                if acct-status = 9
                        process-sw = 1
                        record-count = i-sub - 1
                else
                        error-msg = "problem reading the file; program aborting – press enter"
                        go to end-program
                end-if
        end-if
sort-array: process-sw = 0
        t-sub = i-sub
        perform match-ssn until process-sw = 1
match-ssn: t-sub = t-sub + 1
        if c-element (i-sub)(86:5) > c-element(t-sub)(86:5)
                hold-line = c-element (i-sub)
                c-element (i-sub) = c-element (t-sub)
                c-element (t-sub) = hold-line
        end-if
        if t-sub = record-count
                process-sw = 1
        end-if
write-file: new-record = c-element(i-sub)
        write newfile
        if new-status > 0
                error-msg = "problem writing the file; program aborting – press enter"
                go to end-program
        end-if
end-program: screen(24,20) error-msg input
        end
```

The paragraph
load-array
should be recognizable as logic we've done before, just reading in a file of data and loading it to an array. We accomplish the read of the first record because

account-number

was initially set to

9.

Assuming the read is without fail, the first account record is moved to the first element of our array. The variable

i-sub

will be 1 to accomplish this but when we read the second record, it will be 2 and thus we will move that record to the second element of our array. This will continue until we reach the end of the file, which is found when

acct-status

is equal to 9. The lines

```
if acct-status = 9
        process-sw = 1
        record-count = i-sub - 1
else
        error-msg = "problem reading the file; program aborting – press enter"
        go to end-program
end-if
```

take care of that and note that we set

process-sw

to 1 so that our perform of the paragraph will end. One thing we don't want is the program to keep looping and never end. We also set the variable

record-count

to 1 less than

i-sub.

We have to subtract that 1 because the variable

i-sub

at this point is 1 more than the number of records in the file since it will be in this paragraph once more after the last record is read. We will use

record-count

later as we sort the table and also when we write out the records to a new file from our sorted table. The final

else

will result if

acct-status

is neither 0 nor 9 and that means that there is something drastically wrong with our file. This shouldn't happen but we put this code into our program just in case since there will be times when problems can occur that are out of our control. That takes care of the reading of our file and loading to the array.

We will come back to the actual sort in a while but next let us consider

```
write-file: new-record = c-element(i-sub)
```

```
write newfile
if new-status > 0
        error-msg = "problem writing the file; program aborting – press enter"
        go to end-program
end-if
```

which will take the sorted array and write it out to a new file,

newfile

one record at a time. This process can be described as a sequential write, since one record after another is being written to the output file. It's a bit different from our earlier writes. The perform is done by varying

i-sub

from 1 by 1 until it is equal to

record-count.

Thus

i-sub

will start out as 1 and the first element of the table will be written to the file. This process will continue until all the records are written to the file. Thus the loop will continue and if

record-count

were 8, the paragraph would been completed for

i-sub

equal to 8 and that would end the

perform

since

i-sub

would equal

record-count.

There is one note of caution that I must add. Some systems would not write out the last record using our specific logic. We would have to say

perform write-file **varying** i-sub **from** 1 **by** 1 **until** i-sub > record-count

in the perform statement. That depends on when the variable

i-sub

is checked against

record-count.

In our system it is done after the last line of the paragraph and so we did write out the last record. However, other systems might check before the paragraph starts. You can see that if that is the case, the very last record would not be written since the comparison of

i-sub

to

record-count

would result in an equal condition and the perform would be done without writing out the last record. This is a small point but something that you have to consider. If you write out records to a file and somehow the last record is missing, this could be your problem.

If there is a problem with the

write
of the record, the
new-status
will have a value of something other than 0. We check this field to make sure that the write didn't have a problem, which it really shouldn't. However, by doing a check on this status field, if a problem does occur, we have a message to tell us that somehow there was a problem. If we omitted the check and had an abend, someone would have to research the results. By our process we know right away that we had a problem, saving time and frustration.

With the read of the input file and the load of the array, we can now concentrate on the sort. The sort here is referred to as a bubble sort. Just as bubbles float upward, we shall bring the smallest element to the top and the largest will wind up at the bottom. In this case, we'll look at the zip code and wind up with the smallest zip code at the top and the largest zip code at the bottom. All the zip codes will be sorted the way we expect them to be. For sorting on this field, we'll start with the first one and compare it to the second. For example, if the first zip code is the same or less than the second, we'll do nothing. Had it been reversed, we would have swapped the entire first record of data with the data of the second. Note that we are not swapping only zip codes. In either case, the record with the smallest zip code will now be in the first position in the table. We will now repeat the process comparing the zip codes of the first record with the third one. After this compare and swap – if necessary – the record corresponding to the smaller zip code will be in the first position of the table.

We now need to repeat this compare process using the first and fourth records, then the first and the fifth ones and so on until our last compare will be between the first record's zip code and the last record's zip code. This checking is occurring within our array. When we finish this first step of the process, we will without any doubt have the record with the smallest zip code in the first position of the array.

We'll now go on and repeat the same process but this time we will start with the second record and the third – comparing zip codes – then the second and the fourth and so on just as before. Whenever we have to, we will swap rows of the array just as we did earlier – actually swapping records. When we finish this pass and wind up comparing the second zip code to the last one, we will have the record with the smallest zip code in the first position of the array and then the record with next larger one in the second position of the table.

You can probably guess what the next step will be. We will proceed with the third record, then the fourth one and so on until we start with the second last record and compare its zip code to the zip code of the last record, and we shall be done. At this point our array is completely sorted from smallest zip code to the largest. There probably will be two records with the same zip code, but in our case, that won't matter. There is just one word of caution. In order to do a swap of two lines of the array, we can't just move the first to the second and then the second to the first. If we did that, we would have lost the second line. We will need a temporary place for either line and then three moves will be required. One way would be to move the first line to the temporary line, the second to

the first and finally the temporary line back to the second. In this way we will have achieved the swap without losing any line.

Again, I can't emphasize enough that even though we are comparing zip codes, our array consists of more data than just that field. We need to swap the entire line and not just parts of it as would be done if we were to merely swap the zip codes. After all, if we did that Chris Smith could wind up with Pat Jones' zip code and we don't want that.

With that in mind the lines

```
perform sort-array varying i-sub from 1 by 1 until i-sub = record-count - 1
sort-array: process-sw = 0
t-sub = i-sub
perform match-ssn until process-sw = 1
match-ssn: t-sub = t-sub + 1
if c-element (i-sub)(86:5) > c-element(t-sub)(86:5) then
        hold-line = c-element (i-sub)
        c-element (i-sub) = c-element (t-sub)
        c-element (t-sub) = hold-line
end-if
if t-sub = record-count
        process-sw = 1
end-if
```

will do just exactly what we need to do to sort the array. Recall that

```
record-count
```

represents the number of records in the table. The main perform will loop through varying

```
i-sub
```

from 1 until it is 1 less than the record count. Thus if we had 10 records, the

```
sort-table
```

procedure would represent 9 groups of compares, the first number against the remaining 9, then the second number against the remaining 8 and so on down the line. These groups of compares will be done by the procedure

```
match-ssn.
```

Note that

```
t-sub
```

will be equal to

```
i-sub
```

to start when we get to this procedure but very soon it will be 1 larger, then 2 larger and so on. The very first compare will then compare the first number to the second since

```
i-sub
```

will be 1 and

```
t-sub
```

will be 2. The lines

```
if c-element (i-sub)(86:5) > c-element(t-sub)(86:5)
        hold-line = c-element (i-sub)
        c-element (i-sub) = c-element (t-sub)
        c-element (t-sub) = hold-line
end-if
```

compare the zip codes since they are in position 86 of the each record for a length of 5 and do a swap, if necessary. You can see that

```
hold-line
```

is the variable used to accomplish the swap. As a refresher remember that

```
c-element (i-sub)(10:18)
```

represents the part of one element of the table starting in position 10 for a length of 18, depending on the variable

```
i-sub.
```

That isn't the zip code here, but the last name. When

```
t-sub = record-count,
```

this means we just compared the first zip code in the table to the last one and thus we're done with that group of compares. We next set

```
process-sw
```

to 0 again and proceed as we outlined until our sort is complete.

Naturally you won't need to write a sort like this since you should have easier ways of doing this process. But if you have to, you could write your own sort. This is the way it was done in the pioneer days of computing. Actually there were a great many more things you had to do that we take for granted today.

And now for that question in chapter 9: Could a file be read sequentially if we began at the last record and read the entire file backwards with the first record on the file being the last record read. In that case, would you get a *begin of file* rather than an *end of file*? Seriously, if the file we're talking about is the account number file, just sort it by the account number in descending order. Take the result and read it sequentially and you'll have the account number file read backwards. You don't have to use the bubble sort as easier ways to get the same result.

24. A program in action

We spent a great deal of time on concepts of programming. Before continuing, let's present an easier way to sort a file than by the bubble sort of the last chapter. In this case, we want to sort the account number file in ascending order by zip code, then by last name, first name and middle initial, producing a report with all those fields and the account number.

```
program-name: sortedlist
define  main-heading structure
        print-month character(2)
        field character value "/"
        print-day character(2)
        field character value "/"
        print-year character(42)
        field character(74) value "Sorted account number file report"
        field character(5) value "Page"
        page-number integer(3)
define  sub-heading structure
        field character(13) value spaces
        field character(19) value "zip code"
        field character(32) value "last name"
        field character(29) value "first name"
        field character(15) value "initial"
        field character(22) value "account number"
define  print-line structure
        field character(13) value spaces
        print-zip-code integer(5)
        field character(14) value spaces
        print-last-name character(18)
        field character(14) value spaces
        print-first-name character(15)
        field character(14) value spaces
        print-middle-initial character
        field character(14) value spaces
        print-account-number integer(9)
define file acctfile record account-record status acct-status structure
        field character(100)
define file sortfile record sorted-record status sort-status key account-number structure
        account-number integer(9)
        last-name character(18)
        first-name character(15)
        middle-initial character
        field character(42)
```

```
        zip-code integer(5)
        field character(10)
define work-date character(8)
define record-counter integer(5) value 0
define page-counter integer(3) value 0
define line-counter integer(2) value 54
define error-msg character(60) value spaces
work-date = date
print-month = work-date(5:2)
print-day = work-date(7:2)
print-year = work-date(3:2)
sort acctfile ascending zip-code last-name first-name middle-initial into sorted-record
if sort-status > 0
        error-msg = "sort problem – program ending"
        go to end-program
end-if
account-number = 0
read-file: readnext sortfile
        if sort-status = 0
                record-counter = record-counter + 1
                print-account-number = account-number
                print-last-name = last-name
                print-first-name = first-name
                print-middle-initial =  middle-initial
                print-zip-code = zip-code
                line-counter = line-counter + 1
                if line-counter > 54
                        perform print-headings
                end-if
                print print-line
                go to read-file
        else
                if acct-status not = 9
                        error-msg = "There was a problem with the account file – program ending"

                end-if
        end-if
        go to end-program
print-headings: page-counter = page-counter + 1
        page-number = page-counter
        line-counter = 5
        print page main-heading
        print skip(2) sub-heading
        print skip
```

```
end-program: print skip(2)
        print error-msg
        end
```

You will note that this is a great deal easier than the bubble sort records we used in the last chapter. I also made a few other modifications to handle sort or read problems. A new keyword here is what you might expect,

sort.

You might say it magically sorts the account number file in ascending order by the four fields, zip code, last name, first name and middle initial – in that order – into a new file,

sortfile.

Each of these two lines,

define file acctfile **record** account-record **status** acct-status **structure**

and

define file sortfile **record** sorted-record **status** sort-status **key** account-number **structure**

spells out the record layout, status and structure, enabling the

sort

to work in the proper manner. The keyword,

ascending

leaves the records sorted from lowest order to highest by those four fields in the way requested. To obtain the data starting from the bottom of the alphabet, you would use the keyword,

descending.

From the sorted file, we just produce the report as before. Since there could be problems with the sort or the read, we have provided for each of those, which you can see at the sort and read procedures respectively. If there is a problem with the printing of the report, it's time for a new computer. Are you glad that we don't need that bubble sort?

Now let us see how we can take advantage of these ideas in the real world. We deal with a program or two that can be helpful in ordering goods in the grocery store, with little intervention on the part of any human being. We will not actually do the programming but merely indicate what could be done to have orders for stock automatically placed.

The front of the store already has an automated system for pricing of groceries and producing totals for the customer to pay for the goods purchased. This involves scanning the product to get a price for the item. The system works well but we shall now add an online program to the process. This will trigger ordering of cases of soup, pickles, cereal or whatever else is needed to guarantee that we don't run out of stock for a particular product. To illustrate what will be done, I will deal with one product alone, dill pickle relish. Of course the procedure will work for any item in the store.

When the jar of relish is scanned at the register, besides the normal routine, an online program will be running which will add 1 (or 2 if that many jars of the relish are purchased) to a counter. This counter represents the number of jars of dill pickle relish that could be added to completely fill the space on the shelf that this product occupies.

Besides this counter, the product code for this relish has a few other fields in the record on a file that we need. One is the number of jars in a case of dill pickle relish. Another is the number of jars that will fit on the shelf in the store in the assigned area in the pickle aisle.

There could be other fields as well such as the description of the product. However, I think you will agree that we really don't need that specific field to accomplish what needs to be done regarding ordering. For argument sake let us say that one case of relish holds 12 jars and the shelf has room for 30 jars of the condiment. If the counter is currently at 11, when one jar of dill relish is scanned at any checkout, the counter will be incremented to 12 and this will cause an action to be taken which will update an order file and the counter reset. One case of this relish can be ordered, so it is.

The order file will be an indexed file with the record key being the product code. At the time when that twelfth jar is scanned – triggering an order for another case of relish – the order file will be read for that specific code and if a record is found, that meant that at least one case of relish was ready to be ordered. Now another one is needed, so the number of cases will be bumped up by 1. That order record will be rewritten. A read of the order file resulting in no record being found means that only one case is ordered.

This will work for every product in the store and as any item is scanned, we may not write out any record to the order file but at least we will increment the counter on the file for that product. Eventually this file of orders will be transmitted to the warehouse and that probably will happen at the end of the day or maybe early the next morning. Of course it could happen twice a day or maybe only every other day. Once the transmission takes place, the order file will be cleared since we don't need to have it duplicated. Before the order is transmitted, it will be backed up. This is done just in case there is a problem with the sending of the file to the warehouse.

You might wonder about the case where a customer somehow knocks a bottle of relish off the shelf and it winds up all over the floor of the pickle aisle. In that case the item will not be put through the scanner at one of the front registers but there will be one less bottle of relish on the shelf. This scenario could mess up the whole order process as a case of stock could fit on the shelf but the counter for that product would be at one less than the number of items in a case. To remedy this problem, we will have a scanner in the back room just for damaged goods. Some grocery clerk will have to get a bottle of the relish off the shelf and run it through the scanner for every broken bottle of relish. He or she also has to eventually put that jar back.

This will solve the reconciliation problem and at the end of the day you may even get a printout of all the damage for the day as well as the amount of the loss. But there is another potential problem having to do with shoplifting. If someone swipes one bottle of relish and it winds up in the car without going through the scanner up front, we have the same problem but no apparent way of reconciling it. Fortunately there is technology available where a product will not be able to get out the door without sounding an alarm. This could be what we need for stopping thieves. Another option is to have the ability to have an explosion triggered by the bottle leaving the store without being scanned. That option involves cleanup, however, and is not practical.

We may also find another solution to shoplifting so as to not mess up our ordering system but what about in-store thieves? Specifically what I am referring to is the bag of chips, which accidentally winds up as *damaged* in order to satisfy the salt cravings of the stock clerks. One rule of thumb is that a product found on the wrong shelf is fair game for starving grocery store workers. Maybe you've heard of it. This shouldn't happen to a bottle of relish. Once again someone is going to have to run the bag of chips through the backroom scanner. In this way appetites will be satiated and we don't have to worry about messing up the count. I am sure there are other scenarios that need consideration but there are probably ways to solve them as well.

Thus all the stock in the store can be ordered through the computer although it will still take manual intervention to fill the order at the warehouse, deliver it, unload it at the supermarket and put it on the shelf. Certain tasks will be eliminated but some will remain. In addition the system of keeping track of the stock can be used in a myriad of other ways. We can track how many cases of relish sell in a day, week, month, or during a specific season. When I worked in a supermarket I ordered stock for the glass aisle, which held soups, pickles and relishes. Now you know why I talked about dill pickle relish. At that time, we had averages for each product indicating how much would sell in a week so that we could order accordingly. So if the shelf held 3 cases of a particular item but 4 cases would be sold on an average week, we would order the latter amount for the week.

With our system there is no need to do that, as the system will order as needed. I recall my ordering days when the grocery manager used to order and we wound up with plenty of overstock in the back room. Apparently he used to dream of scenarios where the consumer would buy more than the average for some item. Obviously that didn't happen and hence the surplus. I myself did my best to keep the overstock in the back room to a minimum even if the weekend found some items wanting. You could always order more next week.

Getting back to our automated system, note that we won't have overstock and that means that we won't need all that space in the back room. When a load comes in from the warehouse, we can unload it off the truck and take it directly to the aisles for stocking. This assumes that we have the manpower to do it but that is only a simple scheduling of help problem. Our system is turning out to be quite beneficial in making profits.

Another good thing about our system is all the information available. Getting back to our relish, suppose we notice that it sells two cases a week and the pickled watermelon rind next to it sells half that amount. At the same time the space for the rind holds more bottles that the dill relish. What the information can allow us to do is to change the allotment on the shelf for the two products, giving more space to the relish since it sells more each week. This will mean that we will be less likely to run out relish on the shelf since we can store more. This will also make the manager happier.

Obviously the computer can eliminate work and that means those laboring in the store won't have to do certain things. These tasks could even be assignments that people take pleasure and pride in. At the same time the information available will not only keep stock clerks working, it will also allow them to be more productive and creative. They may have less of a physical challenge, but they will be required to perform in a cerebral way. When they get that bonus at the end of the year, they can say that they earned it.

That concludes our short trip into the basic ideas of computer programs and how they work. This is a only a very tiny piece of the world of computers as there is no mention of the architecture of a computer or how any of the keywords indicating action really work inside the computer. We just take it for granted that they do, but as we have seen, sometimes matters get fuzzy and complicated. It may be up to us to clear up some of the haze. Unfortunately, that may not be an easy matter. We need only do the best we can, remembering that computers are machines that are programmed by people. When rocket scientists get involved, who knows what could happen. Happy computing!

APPENDIX - KEYWORDS

ascending – used to sort one or more fields in a file from smallest to largest order

by – used for the increment in a perform statement

call – for processing through another program

character – a data type that includes just about everything

copy – for bringing in common code into a program

date – the value of today's date in yyyymmdd format

decimal – a data type for numbers

define – for describing files, their composites and other variables

delete – removes a record from a file

descending – used to sort one or more fields in a file from largest to smallest order

element – breakdown of pieces of fields in an array

else – the option resulting from an if statement

end – to terminate a program

end-if – termination of an if statement

erase – to remove old data from the screen

field – used to represent some data, more or less fixed

file – relates to a group of records of data

found – points to the record number from a match in a table from a search

from – used as a starting point in perform statements

go to – used for branching

if – used for making a decision

index – for checking the occurrence of characters within a string

input – allows data from the user into the program through the screen

integer – a data type that represents whole numbers

into – for moving data from one place to another

key – how an indexed file is read

link – enables data to be passed between programs

mask – for editing output that is displayed or printed

mod – gives the remainder after division

occurs – for describing the makeup of data in an array

page – will start printing at the top

pause – delays activity in processing

perform – allows a procedure to be processed

print – allows output to a report

program-name – to differentiate one program from another

read – allows access to a file by obtaining a specific record

readnext – allows sequential file access procuring one record after another

record – indicates a part of a file

screen – gets results on a monitor

search – allows looking through a table for a match

signed – allows negative as well as positive values

skip – for printing blank lines on a report

sort – used to order a file by one or more fields

source – points to a library for dynamic data

space – one or more blanks

spaces – can be used instead of space

status – used for checking success of file accesses

structure – to breakdown a record into parts

table – an array of data with similar characteristics

times – used with the occurs clause for an array

until – controls processing in a perform

update – locks a record in a file for modification

using – for data transferred between programs through a call

value – for initializing a field

varying – allows incrementation of a counter by a set amount

write – creates an output record

OPERATORS

=	–	used to assign values as well as to compare fields
+	–	addition
-	–	subtraction
>	–	greater than
<	–	less than
not =	–	not equal
<=	–	less than or equal
>=	–	greater than or equal
*	–	multiplication
/	–	division

STOP DREAMING AND BECOME AN AUTHOR YOURSELF TODAY!

It's Free, Easy and Fun!

At our sister website, Foboko.com, we provide you with a free 'Social Publishing Wizard' which guides you every step of the eBook creation/writing process and let's your friends or the entire community help along the way!

LOGON ONTO FOBOKO.COM

and get your story told!

FOBOKO.COM