PRINCIPLES OF COMPUTER PROGRAMMING

Kern W. Dickman

Many years ago an advertisement appeared frequently in popular magazines which displayed a photograph of a man or a woman seated before a piano. The caption below read: "I learned to play in five easy lessons." We are going to learn the principles of computer programming in one easy lesson.

We know, of course, that it is possible to learn to play a simple tune, "Twinkle, Twinkle, Little Star," and a few others, over a period of five weeks, with practice and five easy lessons. This is, however, a limited repertoire. This hoax—that one can learn anything in five easy lessons—will be perpetuated by this talk. A few simple, but typical, procedures will be examined, and the corresponding computer programs will be outlined. The characteristics of programs in connection with these exercises will be discussed, but it takes years to learn to perform well on a computer.

Let us begin with a simple, but typical, computer program: the calculation of a mean or average. We may wish to calculate a student's grade point average, or the average daily amount of money spent on food, or, for income tax purposes, the average number of miles traveled in a car, or, for the library, the average cost of cataloging a book.

The mean, of course, is found by adding the scores together and then by dividing the sum by the number of observations, N, as shown in equation (1).

$$(1)\quad \text{MEAN} = \frac{1}{N}(X_1 + X_2 + \ldots . X_N) = \frac{1}{N}\Sigma X_i$$

To add to the clarity of the example, the variable X shall be replaced with some real numbers as in equation (2).

$$(2)\quad \text{MEAN} = \frac{1}{4}(-16 + 20 - 4 + 8) = 2$$

Kern W. Dickman is Assistant Director of the Statistical Service Unit, University of Illinois, Urbana, Illinois.

A characteristic of an equation is that it is in balance. One side of the equation is equal to the other. Therefore, we may regard an equation as a static statement of a relationship. Moreover, if each of several persons were asked to evaluate the right side of equation (2), a great variety of procedures would emerge.

Some persons first would add the positive numbers, then the negative numbers, and finally the difference would be found and divided by the number of observations. Others would hunt for easy number combinations. In equation (2) they might notice that -16, +20, and -4 add to zero. So +8 would be divided by +4. Many would begin by rewriting the numbers in a column, or in two columns, one for positive numbers and one for negative numbers. It is also possible to begin by dividing each number in the sum by +4. Then the sum of the quotients is equal to the mean.

An equation is a static relationship, for it does not tell us what steps to take or even that we should take the steps. A computer program, on the other hand, is a set of dynamic statements which imply action. It tells a computer to perform a series of steps, and it specifies exactly in what order these steps are to be performed.

Let us introduce at this time a dynamic relational symbol to replace the equal sign. This symbol, : =, has been borrowed from ALGOL, a computer programming language. It means more than "equal"; it means "set equal to." Thus, the statement, SUM : = 0, means set SUM equal to zero. Notice that this relational symbol permits such statements as

$$(3)\quad I := I + 1.$$

As an equation, (3) is nonsense, for we can transpose I from one side of the equation to another, yielding 0 = 1. As a dynamic statement of relationship, however, (3) makes sense, for it means that I is to be set equal to the previous I plus one.

Now let us suppose that we have some hypothetical computer. Like all computers, our computer has an arithmetic unit capable of performing addition, subtraction, multiplication, and division, and other logical operations which shall be introduced later as needed.

Like all computers, our computer has a memory consisting of a large number of locations or slots in which information can be stored. These locations are numbered from zero through L, and they can be referred to by these address numbers. In our computer, each of these locations can also be referred to by a symbolic address. Thus, SUM might be the symbolic address of one such location. However, we can call them anything we like: SUM, MEAN, CHARLIE, ADA, or BOOK. In fact, we can refer to a series of them as BOOK (1), BOOK (2), BOOK (3), and so on. Such a list shall be used in our program to calculate a mean. The scores will be referred to as X (1), X (2), X (3), and X (4), or more generally, as X (I) where I

Address		Contents
0		
1		
2		
:		
:		
	N	4
	I	
	SUM	
	MEAN	
:		
:		
	X (1)	-16
	X (2)	20
	X (3)	- 4
	X (4)	8
:		
:		
L - 1		
L		

Fig. 1—Memory Locations

can take on successively increasing values (see Fig. 1). For our program we shall need a location to contain the sum as we add the numbers together. We shall refer to it symbolically as SUM. We shall want to keep track of the numbers as we add them by increasing our tally location I so that exactly N = 4 numbers are processed.

Since computers can be instructed to perform almost an infinite number and variety of tasks, there is a correspondingly large variation in computer programs. A characteristic of nearly all programs, except those of the most trivial kind, is that they are made up of one or more iterative loops. Any sequence of computer instructions which is operated upon repeatedly may be called an iterative loop. Our program to calculate a mean, which we shall begin to assemble shortly, has only one such loop. It will serve, nevertheless, to illustrate the properties of iterative loops.

It is often necessary to set up initial conditions prior to entry into an iterative loop. In our program, for example, we shall want both locations, SUM and I, to be cleared of the results of any previous computations. Otherwise, our program may miscalculate, and we shall have a program error. Let us use the word initialize to refer to this part of a computer program. For our program, the initialize section will consist of two statements:

```
SUM : = 0
  I : = 0
```

The iterative loop for our program will consist of three statements. Let us label the first statement (L) so that we can go back to it for each iteration. First, we shall write the loop, and then we shall discuss it.

```
(L)  I    : = I     + 1          Increment
     SUM  : = SUM + X (I)        Recurrence relation
     IF   I < N, GO TO (L)       Test
```

The first statement of loop (L) sets I equal to the previous I and tallies one. But the previous I was made zero by the initialize section. Therefore, at this point I is merely equal to one. The second statement, which is called a recurrence relation, says to set SUM equal to the previous SUM, which was cleared to zero by the initialization, and to add X (I). Since I is equal to one, this means to add X (1) or -16. Finally, we arrive at the test to determine if we have added N = 4 scores. Notice that we have underlined IF and GO TO. This underlining is done to avoid confusion with a symbolic address which might be called IF or GO TO. The tally I will be less than N, and so we follow the directions of the statement and return to the beginning of loop (L).

Let us examine what will happen on successive passes through the iterative loop. Each time I will be increased by 1, and so each time X (I) will refer to the next score in the list. SUM will take on values as follows:

```
First time:    SUM =   0 + X (1) =  -16
Second time:   SUM = -16 + X (2) = -16 + 20 = 4
Third time:    SUM =   4 + X (3) =   4 -  4 = 0
Fourth time:   SUM =   0 + X (4) =   0 +  8 = 8
```

When I becomes 4, it will no longer be less than N, and so the test will not return control to the beginning of loop (L), but will continue to the next part of the program.

The next part of the program will be another process section, but this time no iterative loop is involved. The process consists merely of dividing SUM by N to find the mean.

Finally, we shall want to print the result, so we shall add an instruction to our list of permissible ones. Our hypothetical computer will interpret WRITE JOE as a command to activate the printer by printing the contents of memory location JOE, or whatever other location is specified.

Similarly, we shall need to add a READ instruction to our list to input the list of scores. This will have the form, READ N, X (I), and will be interpreted to mean that N scores will be read and placed at symbolic locations X (1) through X (N).

At this point we can assemble the complete program to calculate a mean.

Program No. 1

	Instructions	Comments
	START	Initialize
	READ N, X(I)	
	SUM : = 0	
	I : = 0	
(L)	I : = I + 1	Process; iterative loop
	SUM : = SUM + X (I)	Increment
	IF I < N, GO TO (L)	Recurrence relation
		Test
	MEAN : = SUM / N	Process
	WRITE MEAN	
	STOP	

You will notice the underlining to distinguish instructions from symbolic addresses. Also the label (L) is enclosed in parentheses for the same reason.

Program No. 1 is a set of dynamic statements which specify exactly how to calculate the mean. Furthermore, it is a perfectly general program, for it will operate as successfully when N = 10,000 or 100,000 as when N = 4.

Notice how compact the program is presented despite the many thousands of operations the program may imply if N were to be large. Notice also how easy it has been to write the program using such a language. Of course, the language we have been using does not exist in any well-developed form, for we have been making it up as we needed it. However, there are such kinds of languages. Examples which are well known are FORTRAN, COBOL, and ALGOL.

Before our program to calculate a mean could be used on a real computer, it would be necessary, of course, to translate these symbolic instructions into machine language. Fortunately, programs exist which do the translation for you, and FORTRAN, COBOL, and ALGOL compilers are now available on most machines.

Let us return to our program, however, for it should be pointed out that it is easy to make mistakes when writing a program. For example, in the iterative loop, the increment statement preceded the recurrence relation. If these were to be interchanged,

```
SUM : = SUM + X (I)
I   : = I   + 1
```

then the numbers to be processed would run from X (0) through X (N - 1). We do not know what number is contained in location X (0). It may cause only a trivial difference in the result, and especially if the list were a long one, the result may appear to be quite reasonable. The program would be accepted as correct, and

others would begin to use the program. Such an error could remain undetected for a long time. The consequences could be serious if crucial decisions were made on the basis of the results. In any event, it is embarrassing when such results are published.

Let us also consider an effect from writing the wrong test statement. Suppose that this statement is

$$\underline{\text{IF}}\ I \leqq N,\ \underline{\text{GO}}\ \underline{\text{TO}}\ (L).$$

This would result in an extra passage through the loop and so $X(N+1)$ would be added to the sum. Again the error may or may not be readily detected depending upon what might be contained at this location. Of course, there are many other ways to make errors, but the point in these illustrations is to show how easy it is to make an error. The warning is always to check the program carefully.

If you think that this warning is an overemphasis, let me remind you that some years ago, when the United States was trying to close the missile gap, a missile blew up at a cost of 40 millions of taxpayer dollars as a result of a trivial programming error. A hyphen was omitted from the program.

There are many ways to check a program, but one way is actually to trace the steps in the same order as they will occur during the execution of the program. To save time, let us illustrate this by checking the iterative loop of our program only, as illustrated in Table 1.

TABLE 1

Checking the Loop

Steps	I	SUM	X(I)	Test	Comments
	0	0	?		Initial conditions
1	1		-16		1st iteration
2		-16			
3				1<4	Yes, return to (L)
4	2		20		2nd iteration
5		4			
6				2<4	Yes, return to (L)
7	3		- 4		3rd iteration
8		0			
9				3<4	Yes, return to (L)
10	4		8		4th iteration
11		8			
12				4<4	NO, continue

It appears from Table 1 that the result + 8 at the end of the loop is correct, and this confirms that the program also is correct.

One reason that we have spent a large part of our time in discussing a program to calculate a mean is that it has served as a vehicle to introduce concepts and principles used in programming.

However, it will also serve as an outline for a large number of other procedures. For example, by modifying the recurrence relation we could use the program to find the mean of differences, or mean of products, or mean of quotients. Similar iterative loops are quite typical of other programs, and complex programs often consist of the assembling of a series of similar iterative loops. Thus, we have examined in detail a basic building block typical of many larger programs.

Before we end our discussion of programming, we should examine a program of a different kind—one involving more decisions. To illustrate this variety of program, let us pose a simple problem.

We are given a set of N observations, X (I), which may be the scores of students, the costs of books, or the weights of hogs. These also might be the titles of books since, as far as a computer is concerned, letters are merely special numbers. We wish to locate and print the largest number in the set, call it L, and the smallest number in the set, call it S.

Although this may appear to be a somewhat trivial exercise, like the mean, it is related to other more important problems. It is not unusual to want to place a set of numbers in order of size, or to alphabetize a set of book titles. To do either, it may be necessary to continue to locate the next smaller or next larger number.

We shall not repeat our step-by-step development as was done for the calculation of the mean, for by this time we know the main ideas connected with writing a program. Instead let us begin immediately to write our program. Later we shall test it to ascertain if there are errors.

Program No. 2

Instructions	Comments
START	Initialize
READ N, X(I)	
	At the point where only one number has
L : = X(1)	been examined, it is both the largest and
S : = X (1)	and the smallest.
I : = 2	The next number to be examined will be X(2).
(A) IF X(I) > L, GO TO (C)	Test if larger than L.
IF X(I) < S, GO TO (D)	If not, test if smaller than S.
(B) I : = I + 1	If neither, increment and test for end
IF I ≤ N, GO TO (A)	of loop.
WRITE S	If end of loop, process.
WRITE SPACE	We do not want to print L on top of S,

WRITE L	so we cause a carriage advance on the
STOP	printer.
(C) L : = X(I)	If X(I) > L, replace L with X(I).
GO TO (B)	Return to loop.
(D) S : = X(I)	If X(I) < S, replace S with X(I).
GO TO (B)	Return to loop.

Notice that this program is longer than the program for the mean despite the fact that no calculations are involved. The complexity of a program is more closely related to the number of decisions that it is required to make than to the number of calculations.

Further, errors are more likely to be made in problems with a large number of decisions so we shall check our program by tracing the steps one-by-one with a set of four numbers, as follows:

$$X(1) = +2,\ X(2) = -2,\ X(3) = +4,\ X(4) = +1$$

TABLE 2

Checking Program 2

Steps	I	X(I)	S	L	Tests and comments
					Initialize
1					Read X(I)
2	1	+2		+2	First number is both S and L
3			+2		
4	2	-2			
					Process loop
5					Is $-2 > L$? No, continue
6					Is $-2 < S$? Yes, replace
7			-2		
8	3	+4			Increment
9					Test for end: Is $3 \leqq 4$? Yes, return to (A)
10					Is $+4 > L$? Yes, replace
11				+4	
12	4	+1			Increment
13					Test for end: Is $4 \leqq 4$? Yes, return to (A)
14					Is $+1 > L$? No, continue
15					Is $+1 < S$? No, continue
16	5	?			Increment
17					Test for end: Is $5 \leqq 4$? No, continue End of loop; next process
18					WRITE S = -2
19					Space
20					WRITE L = +4

It appears that our program to find the largest and smallest number of a set is correct, for we have found the correct results.

It has often been remarked that one learns more by teaching a course than by taking a course. It also seems to be true that one learns more in trying to develop a program for a set of operations than he would either by teaching these procedures to others or by being taught.

Now that we have learned to write programs in one easy lesson, we can apply our talents to automating certain library applications. However, at the beginning of this paper, you were warned that it takes time to develop into an experienced and proficient programmer. Indeed, it has been said that an experienced programmer is one who has already made all of the mistakes. In that sense, we are still amateurs, for we wrote two programs without errors.